Proofs, Computability, Undecidability, Complexity, And the Lambda Calculus An Introduction

Jean Gallier and Jocelyn Quaintance
Department of Computer and Information Science
University of Pennsylvania
Philadelphia, PA 19104, USA
e-mail: `jean@cis.upenn.edu`

April 28, 2020

Preface

The main goal of this book is to present a mix of material dealing with

1. Proof systems.
2. Computability and undecidability.
3. The Lambda Calculus.
4. Some aspects of complexity theory.

Historically, the theory of computability and undecidability arose from Hilbert's efforts to completely formalize mathematics and from Gödel's first incompleteness theorem that showed that such a program was doomed to fail. People realized that to carry out both Hilbert's program and Gödel's work it was necessary to define precisely what is the notion of a *computable function* and the notion of a mechanically checkable proof. The first definition given around 1934 was that of the class of computable function in the sense of Herbrand–Gödel–Kleene. The second definition given by Church in 1935-1936 was the notion of a function definable in the λ-calculus. The equivalence of these two definitions was shown by Kleene in 1936. Shortly after in 1936, Turing introduced a third definition, that of a Turing-computable function. Turing proved the equivalence of his definition with the Herbrand–Gödel–Kleene definition in 1937 (his proofs are rather sketchy compared to Kleene's proofs). All these historical papers can be found in a fascinating book edited by Martin Davis [25].

Negative results pointing out limitations of the notion of computability started to appear: Gödel's first (and second) incompleteness result, but also Church's theorem on the undecidability of validity in first-order logic, and Turing's result on the undecidability of the halting problem for Turing machines. Although originally the main focus was on the notion of function, these undecidability results triggered the study of computable and noncomputable sets of natural numbers.

Other definitions of the computable functions were given later. From our point of view, the most important ones are

1. RAM programs and RAM-computable functions by Shepherdson and Sturgis (1963), and anticipated by Post (1944); see Machtey and Young [36].
2. Diophantine-definable sets (Davis–Putnam–Robinson–Matiyasevich); see Davis [8, 9].

We find the RAM-progam model quite attractive because it is a very simplified realistic model of the true architecture of a modern computer. Technically, we also find it more convenient to assign Gödel numbers to RAM programs than assigning Gödel numbers to Turing machines. Every RAM program can be converted to a Turing machine and vice-versa in polynomial time (going from a Turing machine to a RAM is quite horrific), so the two models are equivalent in a strong sense. So from our perspective Turing machines could be dispensed with, but there is a problem. The problem is that the Turing machine model seems more convenient to cope with time or space restrictions, that is, to define complexity classes.

There is actually no difficulty in defining nondeterministic RAM programs and to impose a time restriction on the program counter or a space restriction on the size of registers, but nobody seems to follows this path. This seems unfortunate to us because it appears that it would be easier to justify the fact that certain reductions can be carried out in polynomial time (or space) by writing a RAM program rather than by constructing a Turing machine. Regarding this issue, we are not aware than anyone actually provides Turing machines computing these reductions, even for SAT.

In any case, we will stick to the tradition of using Turing machines when discussing complexity classes.

In addition to presenting the RAM-program model, the Turing machine model, the Herbrand–Gödel–Kleene definition of the computable functions, and showing their equivalence, we provide an introduction to recursion theory (see Chapter 7). In particular, we discuss creative and productive sets (see Rogers [44]). This allows us to cover most of the main undecidability results. These include

1. The undecidability of the halting problem for RAM programs (and Turing machines).
2. Rice's theorem for the computable functions.
3. Rice's extended theorem for the listable sets.
4. A strong form of Gödel' first incompleteness theorem (in terms of creative sets) following Rogers [44].
5. The fact that the true first-order sentences of arithmetic are not even listable (a productive set) following Rogers [44].
6. The undecidability of the Post correspondence problem (PCP) using a proof due to Dana Scott.
7. The undecidability of the validity in first-order logic (Church's theorem), using a proof due to Robert Floyd.
8. The undecidability of Hilbert's tenth problem (the DPRM theorem) following Davis [8].

9. Another strong form of Gödel' first incompleteness theorem, as a consequence of diophantine definability following Davis [8].

The following two topics are rarely covered in books on the theory of computation and undecidability.

In Chapter 6 we introduce Church's λ-calculus and show how the computable functions and the partial computable functions are definable in the λ-calculus, using a method due to Barendregt [4]. We also give a glimpse of the second-order polymorphic λ-calculus of Girard.

In Chapter 8 we discuss the definability of the listable sets in terms of Diophantine equations (zeros of polynomials with integer coefficients) and state the famous result about the undecidability of Hilbert's tenth problem (the DPRM theorem). We follow the masterly exposition of Davis [8, 9].

A possibly unsusual aspect of our book is that we begin with two chapters on mathematical reasoning and logic. Given the origins of the theory of computation and undecidability, we feel that this is very appropriate. We present proof systems in natural deduction style (a la Prawitz), which makes it easy to discuss the special role of the proof–by–contradiction principle, and to introduce intuitionistic logic, which is the result of removing this rule from the set of inference rules. It is also quite natural to explain how proofs in intuitionistic propositional logic are represented by simply-typed λ-terms. Then it is easy to introduce the "Curry–Howard isomorphism." This is a prelude to the introduction of the "pure" (untyped) λ-calculus.

Our treatment of complexity theory is limited to $\mathcal{P}$, $\mathcal{NP}$, co-$\mathcal{NP}$, $\mathcal{EXP}$, $\mathcal{NEXP}$, $\mathcal{PS}$ (PSPACE) and $\mathcal{NPS}$ (NPSPACE) and is fairly standard. However, we prove that SAT is $\mathcal{NP}$-complete by first proving (following Lewis and Papadimitriou [35]) that a bounded tiling problem is $\mathcal{NP}$-complete.

In Chapter 12 we treat the result that primality testing is in $\mathcal{NP}$ in more details than most other sources, relying on an improved version of a theorem of Lucas as discussed in Crandall and Pomerance [6]. The only result that we omit is the existence of primitive roots in $(\mathbb{Z}/p\mathbb{Z})^*$ when p is prime.

In Chapter 13 we prove Savitch's theorem ($\mathcal{PS} = \mathcal{NPS}$). We state the fact that the validity of quantified boolean formulae is $\mathcal{PS}$-complete and provide parts of the proof. We conclude with the beautiful proof of Statman [46] that provability in intuitionistic logic is $\mathcal{PS}$-complete. We not give all the details but we prove the correctness of Statman's amazing translation of a valid QBF into an intuitionistically provable proposition.

We feel strongly that one does not learn mathematics without reading (and struggling through) proofs, so we tried to provide as many proofs as possible. Among some of the omissions, we do not prove the DPRM nor do we show how to construct a Gödel sentence in the proof of the first incompleteness theorem; Rogers [44] leaves this as an exercise! We also do not give a complete proof of Statman's result. However, whenever a proof is omitted, we provide a pointer to a source that contains such a proof.

Acknowledgement: We would like to thank João Sedoc and Marcelo Siqueira, for reporting typos and for helpful comments. I was initiated to the theory of computation and undecidability by my advisor Sheila Greibach who taught me how to do research. My most sincere thanks to Sheila for her teachings and the inspiration she provided. In writing this book we were inspired and sometimes borrowed heavily from the beautiful papers and books written by the following people who have my deepest gratitude: Henk Barendregt, Richard Crandall, Martin Davis, Herbert Enderton, Harvey Friedman, Jean-Yves Girard, John Hopcroft, Bill Howard, Harry Lewis, Zohar Manna, Christos Papadimitriou, Carl Pomerance, Dag Prawitz, Helmut Schwichtenberg, Dana Scott, Rick Statman, Jeff Ullman, Hartley Rogers, and Paul Young. Of course, we must acknowledge Alonzo Church, Gerhard Gentzen, Kurl Gödel, Stephen Kleene and Alan Turing for their extraordinary seminal work.

Contents

Contents 7

1 Mathematical Reasoning And Basic Logic **11**
1.1 Introduction 11
1.2 Logical Connectives, Definitions 12
1.3 Meaning of Implication and Proof Templates for Implication 16
1.4 Proof Trees and Deduction Trees 20
1.5 Proof Templates for $\neg$ 22
1.6 Proof Templates for $\wedge, \vee, \equiv$ 26
1.7 De Morgan Laws and Other Useful Rules of Logic 34
1.8 Formal Versus Informal Proofs; Some Examples 35
1.9 Truth Tables and Truth Value Semantics 39
1.10 Proof Templates for the Quantifiers 42
1.11 Sets and Set Operations 50
1.12 Induction and Well–Ordering Principle 55
1.13 Summary 57
Problems 59

2 Mathematical Reasoning And Logic, A Deeper View **63**
2.1 Introduction 63
2.2 Inference Rules, Deductions, Proof Systems $\mathcal{N}_m^{\Rightarrow}$ and $\mathcal{NG}_m^{\Rightarrow}$ 64
2.3 Proof Rules, Deduction and Proof Trees for Implication 66
2.4 Examples of Proof Trees 72
2.5 A Gentzen-Style System for Natural Deduction 78
2.6 Adding $\wedge$, $\vee$, $\perp$; The Proof Systems $\mathcal{N}_c^{\Rightarrow,\wedge,\vee,\perp}$ and $\mathcal{NG}_c^{\Rightarrow,\wedge,\vee,\perp}$ 80
2.7 Clearing Up Differences Among Rules Involving $\perp$ 90
2.8 De Morgan Laws and Other Rules of Classical Logic 93
2.9 Formal Versus Informal Proofs 96
2.10 Truth Value Semantics for Classical Logic 97
2.11 Kripke Models for Intuitionistic Logic 100
2.12 Decision Procedures, Proof Normalization 103
2.13 The Simply-Typed λ-Calculus 106

2.14 Completeness and Counter-Examples 113
2.15 Adding Quantifiers; Proof Systems $\mathcal{N}_c^{\Rightarrow,\wedge,\vee,\forall,\exists,\perp}$, $\mathcal{NG}_c^{\Rightarrow,\wedge,\vee,\forall,\exists,\perp}$ 115
2.16 First-Order Theories 128
2.17 Basics Concepts of Set Theory 134
2.18 Summary 144
Problems 148

3 RAM Programs, Turing Machines 165
3.1 Partial Functions and RAM Programs 168
3.2 Definition of a Turing Machine 173
3.3 Computations of Turing Machines 175
3.4 Equivalence of RAM programs And Turing Machines 178
3.5 Listable Languages and Computable Languages 179
3.6 A Simple Function Not Known to be Computable 180
3.7 The Primitive Recursive Functions 183
3.8 Primitive Recursive Predicates 191
3.9 The Partial Computable Functions 193

4 Universal RAM Programs and the Halting Problem 199
4.1 Pairing Functions 199
4.2 Equivalence of Alphabets 206
4.3 Coding of RAM Programs; The Halting Problem 208
4.4 Universal RAM Programs 212
4.5 Indexing of RAM Programs 217
4.6 Kleene's T-Predicate 218
4.7 A Non-Computable Function; Busy Beavers 220

5 Elementary Recursive Function Theory 225
5.1 Acceptable Indexings 225
5.2 Undecidable Problems 228
5.3 Reducibility and Rice's Theorem 231
5.4 Listable (Recursively Enumerable) Sets 234
5.5 Reducibility and Complete Sets 240

6 The Lambda-Calculus 245
6.1 Syntax of the Lambda-Calculus 247
6.2 β-Reduction and β-Conversion; the Church–Rosser Theorem 252
6.3 Some Useful Combinators 256
6.4 Representing the Natural Numbers 258
6.5 Fixed-Point Combinators and Recursively Defined Functions 264
6.6 λ-Definability of the Computable Functions 266
6.7 Definability of Functions in Typed Lambda-Calculi 272
6.8 Head Normal-Forms and the Partial Computable Functions 279

7 Recursion Theory; More Advanced Topics **285**
7.1 The Recursion Theorem . . . 285
7.2 Extended Rice Theorem . . . 291
7.3 Creative and Productive Sets; Incompleteness . . . 294

8 Listable and Diophantine Sets; Hilbert's Tenth **301**
8.1 Diophantine Equations; Hilbert's Tenth Problem . . . 301
8.2 Diophantine Sets and Listable Sets . . . 304
8.3 Some Applications of the DPRM Theorem . . . 308
8.4 Gödel's Incompleteness Theorem . . . 311

9 The Post Correspondence Problem; Applications **317**
9.1 The Post Correspondence Problem . . . 317
9.2 Some Undecidability Results for CFG's . . . 323
9.3 More Undecidable Properties of Languages . . . 326
9.4 Undecidability of Validity in First-Order Logic . . . 327

10 Computational Complexity; $\mathcal{P}$ and $\mathcal{NP}$ **331**
10.1 The Class $\mathcal{P}$ . . . 331
10.2 Directed Graphs, Paths . . . 333
10.3 Eulerian Cycles . . . 334
10.4 Hamiltonian Cycles . . . 335
10.5 Propositional Logic and Satisfiability . . . 336
10.6 The Class $\mathcal{NP}$, $\mathcal{NP}$-Completeness . . . 341
10.7 The Bounded Tiling Problem is $\mathcal{NP}$-Complete . . . 347
10.8 The Cook-Levin Theorem . . . 351
10.9 Satisfiability of Arbitrary Propositions and CNF . . . 354

11 Some $\mathcal{NP}$-Complete Problems **359**
11.1 Statements of the Problems . . . 359
11.2 Proofs of $\mathcal{NP}$-Completeness . . . 370
11.3 Succinct Certificates, co$\mathcal{NP}$, and $\mathcal{EXP}$ . . . 387

12 Primality Testing is in $\mathcal{NP}$ **393**
12.1 Prime Numbers and Composite Numbers . . . 393
12.2 Methods for Primality Testing . . . 394
12.3 Modular Arithmetic, the Groups $\mathbb{Z}/n\mathbb{Z}$, $(\mathbb{Z}/n\mathbb{Z})^*$ . . . 397
12.4 The Lucas Theorem . . . 405
12.5 Lucas Trees . . . 408
12.6 Algorithms for Computing Powers Modulo m . . . 411
12.7 PRIMES is in $\mathcal{NP}$ . . . 413

13 Polynomial-Space Complexity; $\mathcal{PS}$ and $\mathcal{NPS}$ **417**
13.1 The Classes $\mathcal{PS}$ (or PSPACE) and $\mathcal{NPS}$ (NPSPACE) 417
13.2 Savitch's Theorem: $\mathcal{PS} = \mathcal{NPS}$ 419
13.3 A Complete Problem for $\mathcal{PS}$: QBF 420
13.4 Provability in Intuitionistic Propositional Logic 428

Bibliography **435**

Symbol Index **439**

Index **441**

Chapter 1

Mathematical Reasoning And Basic Logic

1.1 Introduction

One of the main goals of this book is to show how to

construct and read mathematical proofs.

Why?

1. Computer scientists and engineers write *programs* and build *systems*.
2. It is very important to have *rigorous methods* to check that these programs and systems behave as expected (are *correct*, have *no bugs*).
3. It is also important to have methods to *analyze the complexity* of programs (*time/space complexity*).

More generally, it is crucial to have a firm grasp of the *basic reasoning principles and rules of logic*. This leads to the question:

What is a proof?

There is no short answer to this question. However, it seems fair to say that a proof is some kind of *deduction (derivation)* that proceeds from a set of *hypotheses (premises, axioms)* in order to derive a *conclusion*, using some *proof templates* (also called *logical rules*).

A first important observation is that there are different *degrees of formality* of proofs.

1. Proofs can be very *informal*, using a set of loosely defined logical rules, possibly omitting steps and premises.

2. Proofs can be *completely formal*, using a very clearly defined set of rules and premises. Such proofs are usually processed or produced by programs called *proof checkers* and *theorem provers.*

Thus, a human prover evolves in a *spectrum of formality.*

It should be said that *it is practically impossible to write formal proofs*. This is because it would be extremely tedious and time-consuming to write such proofs and these proofs would be huge and thus, very hard to read.

In principle, it is possible to write formalized proofs and sometimes it is desirable to do so if we want to have absolute confidence in a proof. For example, we would like to be sure that a flight-control system is not buggy so that a plane does not accidentally crash, that a program running a nuclear reactor will not malfunction, or that nuclear missiles will not be fired as a result of a buggy "alarm system."

Thus, it is very important to develop tools to assist us in constructing formal proofs or checking that formal proofs are correct. Such systems do exist, for example Isabelle, COQ, TPS, NUPRL, PVS, Twelf. However, 99.99% of us will not have the time or energy to write formal proofs.

Even if we never write formal proofs, it is important to understand clearly what are the rules of reasoning (proof templates) that we use when we construct informal proofs.

The goal of this chapter is to explain what is a proof and how we construct proofs using various *proof templates* (also known as *proof rules*).

This chapter is an abbreviated and informal version of Chapter 2. It is meant for readers who have never been exposed to a presentation of the rules of mathematical reasoning (the rules for constructing mathematical proofs) and basic logic.

1.2 Logical Connectives, Definitions

In order to define the notion of proof rigorously, we would have to define a formal language in which to express statements very precisely and we would have to set up a proof system in terms of axioms and proof rules (also called inference rules). We do not go into this in this chapter as this would take too much time. Instead, we content ourselves with an intuitive idea of what a statement is and focus on stating as precisely as possible the rules of logic (proof templates) that are used in constructing proofs.

In mathematics and computer science, we **prove statements.** Statements may be *atomic* or *compound*, that is, built up from simpler statements using *logical connectives*, such as *implication* (if–then), *conjunction* (and), *disjunction* (or), *negation* (not), and (existential or universal) *quantifiers*.

As examples of atomic statements, we have:

1. "A student is eager to learn."

2. "A student wants an A."

3. "An odd integer is never 0."

4. "The product of two odd integers is odd."

Atomic statements may also contain "variables" (standing for arbitrary objects). For example

1. human(x): "x is a human."

2. needs-to-drink(x): "x needs to drink."

An example of a compound statement is

$$\text{human}(x) \Rightarrow \text{needs-to-drink}(x).$$

In the above statement, $\Rightarrow$ is the symbol used for logical implication. If we want to assert that every human needs to drink, we can write

$$\forall x(\text{human}(x) \Rightarrow \text{needs-to-drink}(x));$$

this is read: "For every x, if x is a human, then x needs to drink."

If we want to assert that some human needs to drink we write

$$\exists x(\text{human}(x) \Rightarrow \text{needs-to-drink}(x));$$

this is read: "There is some x such that, if x is a human, then x needs to drink."

We often denote statements (also called *propositions* or *(logical) formulae*) using letters, such as A, B, P, Q, and so on, typically upper-case letters (but sometimes Greek letters, φ, ψ, etc.).

Compound statements are defined as follows: if P and Q are statements, then

1. the *conjunction* of P and Q is denoted $P \wedge Q$ (pronounced, P and Q),

2. the *disjunction* of P and Q is denoted $P \vee Q$ (pronounced, P or Q),

3. the *implication* of P and Q is denoted by $P \Rightarrow Q$ (pronounced, if P then Q, or P implies Q).

We also have the atomic statements $\perp$ (*falsity*), think of it as the statement that is false no matter what; and the atomic statement $\top$ (*truth*), think of it as the statement that is always true.

The constant $\perp$ is also called *falsum* or *absurdum*. It is a formalization of the notion of *absurdity* or *inconsistency* (a state in which contradictory facts hold).

Given any proposition P it is convenient to define

4. the *negation* $\neg P$ of P (pronounced, not P) as $P \Rightarrow \perp$. Thus, $\neg P$ (sometimes denoted $\sim P$) is just a shorthand for $P \Rightarrow \perp$, and this is denoted by $\neg P \equiv (P \Rightarrow \perp)$.

The intuitive idea is that $\neg P \equiv (P \Rightarrow \perp)$ is true if and only if P is false. Actually, because we don't know what truth is, it is "safer" to say that $\neg P$ is provable if and only if for every proof of P we can derive a contradiction (namely, $\perp$ is provable). By provable, we mean that a proof can be constructed using some rules that will be described shortly (see Section 1.3).

Whenever necessary to avoid ambiguities, we add matching parentheses: $(P \wedge Q)$, $(P \vee Q)$, $(P \Rightarrow Q)$. For example, $P \vee Q \wedge R$ is ambiguous; it means either $(P \vee (Q \wedge R))$ or $((P \vee Q) \wedge R)$.

Another important logical operator is *equivalence*.

If P and Q are statements, then

5. the *equivalence* of P and Q is denoted $P \equiv Q$ (or $P \Longleftrightarrow Q$); it is an abbreviation for $(P \Rightarrow Q) \wedge (Q \Rightarrow P)$. We often say "$P$ if and only if Q" or even "P iff Q" for $P \equiv Q$.

As a consequence, to prove a logical equivalence $P \equiv Q$, we have to prove **both** implications $P \Rightarrow Q$ and $Q \Rightarrow P$.

The meaning of the logical connectives $(\wedge, \vee, \Rightarrow, \neg, \equiv)$ is intuitively clear. This is certainly the case for *and* $(\wedge)$, since a conjunction $P \wedge Q$ is true if and only if both P and Q are true (if we are not sure what "true" means, replace it by the word "provable"). However, for *or* $(\vee)$, do we mean inclusive or or exclusive or? In the first case, $P \vee Q$ is true if both P and Q are true, but in the second case, $P \vee Q$ is false if both P and Q are true (again, in doubt change "true" to "provable"). We always mean inclusive or.

The situation is worse for *implication* $(\Rightarrow)$. When do we consider that $P \Rightarrow Q$ is true (provable)? The answer is that it depends on the rules! The "classical" answer is that $P \Rightarrow Q$ is false (not provable) if and only if P is true and Q is false. For an alternative view (that of intuitionistic logic), see Chapter 2. In this chapter (and all others except Chapter 2), we adopt the classical view of logic. Since negation $(\neg)$ is defined in terms of implication, in the classical view, $\neg P$ is true if and only if P is false.

The purpose of the *proof rules*, or *proof templates*, is to spell out rules for constructing proofs which reflect, and in fact specify, the meaning of the logical connectives.

Before we present the proof templates it should be said that nothing of much interest can be proven in mathematics if we do not have at our disposal various objects such as numbers, functions, graphs, etc. This brings up the issue of where we begin, what may we assume. In set theory, everything, even the natural numbers, can be built up from the empty set! This is a remarkable construction but it takes a tremendous amout of work. For us, we assume that we know what the set

$$\mathbb{N} = \{0, 1, 2, 3, \ldots\}$$

of *natural numbers* is, as well as the set

$$\mathbb{Z} = \{\ldots, -3, -2, -1, 0, 1, 2, 3, \ldots\}$$

of *integers* (which allows negative natural numbers). We also assume that we know how to add, subtract and multiply (perhaps even divide) integers (as well as some of the basic properties of these operations), and we know what the ordering of the integers is.

The way to introduce new objects in mathematics is to make *definitions*. Basically, a definition characterizes an object by some property. Technically, we define a "gizmo" x by introducing a so-called predicate (or property) gizmo(x), which is an abbreviation for some possibly complicated logical proposition $P(x)$. The idea is that x is a "gizmo" if and only if gizmo(x) holds if and only if $P(x)$ holds. We may write

$$\text{gizmo}(x) \equiv P(x),$$

or

$$\text{gizmo}(x) \stackrel{\text{def}}{\equiv} P(x).$$

Note that gizmo is just a name, but $P(x)$ is a (possibly complex) proposition.

It is also convenient to define properties (also called *predicates*) of one of more objects as abbreviations for possibly complicated logical propositions. In this case, a property $p(x_1, \ldots, x_n)$ of some objects $x_1, \ldots, x_n$ holds if and only if some logical proposition $P(x_1, \ldots, x_n)$ holds. We may write

$$p(x_1, \ldots, x_n) \equiv P(x_1, \ldots, x_n)$$

or

$$p(x_1, \ldots, x_n) \stackrel{\text{def}}{\equiv} P(x_1, \ldots, x_n)$$

Here too, p is just a name, but $P(x_1, \ldots, x_n)$ is a (possibly complex) proposition.

Let us give a few examples of definitions.

Definition 1.1. Given two integers $a, b \in \mathbb{Z}$, we say that *a is a multiple of b* if there is some $c \in \mathbb{Z}$ such that $a = bc$. In this case, we say that *a is divisible by b*, that *b is a divisor of a* (or *b is a factor of a*), and that *b divides a*. We use the notation $b \mid a$.

In Definition 1.1, we define the predicate divisible(a, b) in terms of the proposition $P(a, b)$ given by

$$\text{there is some } c \in \mathbb{N} \text{ such that } \ a = bc.$$

For example, 15 is divisible by 3 since $15 = 3 \cdot 5$. On the other hand, 14 is not divisible by 3.

Definition 1.2. A integer $a \in \mathbb{Z}$ is *even* if it is of the form $a = 2b$ for some $b \in \mathbb{Z}$, *odd* if it is of the form $a = 2b + 1$ for some $b \in \mathbb{Z}$.

In Definition 1.2, the property even(a) of a being even is defined in terms of the predicate $P(a)$ given by

$$\text{there is some } b \in \mathbb{N} \text{ such that } \ a = 2b.$$

The property odd(a) is obtained by changing $a = 2b$ to $a = 2b + 1$ in $P(a)$. The integer 14 is even, and the integer 15 is odd. Beware that we can't assert yet that if an integer is not even then it is odd. Although this is true, this needs to proven and requires induction, which we haven't discussed yet.

Prime numbers play a fundamental role in mathematics. Let us review their definition.

Definition 1.3. A natural number $p \in \mathbb{N}$ is *prime* if $p \geq 2$ and if the only divisors of p are 1 and p.

In the above definition, the property $\text{prime}(p)$ is defined by the predicate $P(p)$ given by

$$p \geq 2, \text{ and for all } q \in \mathbb{N}, \text{ if divisible}(p, q), \text{ then } q = 1 \text{ or } q = p.$$

If we expand the definition of a prime number by replacing the predicate divisible by its defining formula we get a rather complicated formula. Definitions allow us to be more concise.

According to Definition 1.3, the number 1 is not prime even though it is only divisible by 1 and itself (again 1). The reason for not accepting 1 as a prime is not capricious. It has to do with the fact that if we allowed 1 to be a prime, then certain important theorems (such as the unique prime factorization theorem would no longer hold.

Nonprime natural numbers (besides 1) have a special name too.

Definition 1.4. A natural number $a \in \mathbb{N}$ is *composite* if $a = bc$ for some natural numbers b, c with $b, c \geq 2$.

For example, 4, 15, 36 are composite. Note that 1 is neither prime nor a composite.

We are now ready to introduce the proof tempates for implication.

1.3 Meaning of Implication and Proof Templates for Implication

First, it is important to say that there are two types of proofs:

1. *Direct* proofs.
2. *Indirect* proofs.

Indirect proofs use the proof–by–contradiction principle, which will be discussed soon.

Because propositions do not arise from the vacuum but instead are built up from a set of atomic propositions using logical connectives (here, $\Rightarrow$), we assume the existence of an "official set of atomic propositions," or set of *propositional symbols*, $\mathbf{PS} = \{\mathbf{P}_1, \mathbf{P}_2, \mathbf{P}_3, \ldots\}$. So, for example, $\mathbf{P}_1 \Rightarrow \mathbf{P}_2$ and $\mathbf{P}_1 \Rightarrow (\mathbf{P}_2 \Rightarrow \mathbf{P}_1)$ are propositions. Typically, we use uppercase letters such as P, Q, R, S, A, B, C, and so on, to denote arbitrary propositions formed using atoms from $\mathbf{PS}$.

We begin by presenting proof templates to construct direct proofs of implications. An implication $P \Rightarrow Q$ can be understood as an if–then statement; that is, if P is true then Q is also true. *A better interpretation is that any proof of $P \Rightarrow Q$ can be used to construct a proof of Q given any proof of P*. As a consequence of this interpretation, we show later that if $\neg P$ is provable, then $P \Rightarrow Q$ is also provable (instantly) whether or not Q is provable. In such a situation, we often say that $P \Rightarrow Q$ is *vacuously provable*. For example, $(P \wedge \neg P) \Rightarrow Q$ is provable for any arbitrary Q.

It might help to view the action of proving an implication $P \Rightarrow Q$ as the construction of a program that converts a proof of P into a proof of Q. Then, if we supply a proof of P as input to this program (the proof of $P \Rightarrow Q$), it will output a proof of Q. So, if we don't give the right kind of input to this program, for example, a "wrong proof" of P, we should not expect the program to return a proof of Q. However, this does not say that the program is incorrect; the program was designed to do the right thing only if it is given the right kind of input. From this functional point of view (also called constructive), we should not be shocked that the provability of an implication $P \Rightarrow Q$ generally yields no information about the provability of Q.

For a concrete example, say P stands for the statement,

"Our candidate for president wins in Pennsylvania"

and Q stands for

"Our candidate is elected president."

Then, $P \Rightarrow Q$, asserts that *if* our candidate for president wins in Pennsylvania *then* our candidate is elected president.

If $P \Rightarrow Q$ holds, then if indeed our candidate for president wins in Pennsylvania then for sure our candidate will win the presidential election. However, if our candidate does not win in Pennsylvania, we can't predict what will happen. Our candidate may still win the presidential election but he may not.

If our candidate president does not win in Pennsylvania, then the statement $P \Rightarrow Q$ should be regarded as holding, though perhaps uninteresting.

For one more example, let $\mathrm{odd}(n)$ assert that n is an odd natural number and let $Q(n, a, b)$ assert that $a^n + b^n$ is divisible by $a + b$, where a, b are any given natural numbers. By divisible, we mean that we can find some natural number c, so that

$$a^n + b^n = (a + b)c.$$

Then, we claim that the implication $\mathrm{odd}(n) \Rightarrow Q(n, a, b)$ is provable.

As usual, let us assume $\mathrm{odd}(n)$, so that $n = 2k + 1$, where $k = 0, 1, 2, 3, \ldots$. But then, we can easily check that

$$a^{2k+1} + b^{2k+1} = (a + b)\left(\sum_{i=0}^{2k}(-1)^i a^{2k-i} b^i\right),$$

which shows that $a^{2k+1} + b^{2k+1}$ is divisible by $a + b$. Therefore, we proved the implication $\mathrm{odd}(n) \Rightarrow Q(n, a, b)$.

If n is not odd, then the implication $\mathrm{odd}(n) \Rightarrow Q(n, a, b)$ yields no information about the provablity of the statement $Q(n, a, b)$, and that is fine. Indeed, if n is even and $n \geq 2$, then in general, $a^n + b^n$ is not divisible by $a + b$, but this may happen for some special values of n, a, and b, for example: $n = 2$, $a = 2$, $b = 2$.

During the process of constructing a proof, it may be necessary to introduce a list of *hypotheses*, also called *premises* (or *assumptions*), which grows and shrinks during the proof. When a proof is finished, it should have an empty list of premises.

The process of managing the list of premises during a proof is a bit technical. In Chapter 2 we study carefully two methods for managing the list of premises that may appear during a proof. In this chapter we are much more casual about it, which is the usual attitude when we write informal proofs. It suffices to be aware that at certain steps, some premises must be added, and at other special steps, premises must be discarded. We may view this as a process of making certain propositions active or inactive. To make matters clearer, we call the process of constructing a proof using a set of premises a *deduction*, and we reserve the word *proof* for a deduction whose set of premises is empty. Every deduction has a possibly empty list of *premises*, and a single *conclusion*. The list of premises is usually denoted by Γ, and if the conclusion of the deduction is P, we say that we have a *deduction of P from the premises Γ*.

The first proof template allows us to make obvious deductions.

Proof Template 1.1. *(Trivial Deductions)*

If $P_1, \ldots, P_i, \ldots, P_n$ is a list of propositions assumed as premises (where each P_i may occur more than once), then for each P_i, we have a deduction with conclusion P_i.

All other proof templates are of two kinds: introduction rules or elimination rules. The meaning of these words will be explained after stating the next two proof templates.

The second proof template allows the construction of a deduction whose conclusion is an implication $P \Rightarrow Q$.

Proof Template 1.2. *(Implication–Intro)*

Given a list Γ of premises (possibly empty), to obtain a deduction with conclusion $P \Rightarrow Q$, proceed as follows:

1. *Add one or more occurrences of P as additional premises to the list Γ.*

2. *Make a deduction of the conclusion Q, from P and the premises in Γ.*

3. *Delete P from the list of premises.*

The third proof template allows the constructions of a deduction from two other deductions.

Proof Template 1.3. *(Implication–Elim, or Modus–Ponens)*

Given a deduction with conclusion $P \Rightarrow Q$ from a list of premises Γ and a deduction with conclusion P from a list of premises Δ, we obtain a deduction with conclusion Q. The list of premises of this new deduction is the list Γ, Δ.

The modus–ponens proof template formalizes the use of *auxiliary lemmas*, a mechanism that we use all the time in making mathematical proofs. Think of $P \Rightarrow Q$ as a lemma that has already been established and belongs to some database of (useful) lemmas. This lemma says if I can prove P then I can prove Q. Now, suppose that we manage to give a proof of P. It follows from modus–ponens that Q is also provable.

Mathematicians are very fond of modus–ponens because it gives a potential method for proving important results. If Q is an important result and if we manage to build a large catalog of implications $P \Rightarrow Q$, there may be some hope that, some day, P will be proven, in which case Q will also be proven. So, they build large catalogs of implications! This has been going on for the famous problem known as *P versus NP*. So far, no proof of any premise of such an implication involving *P versus NP* has been found (and it may never be found).

Beware, when we deduce that an implication $P \Rightarrow Q$ is provable, we **do not** prove that P **and** Q are provable; we only prove that **if** P is provable **then** Q is provable.

In case you wonder why the words "Intro" and "Elim" occur in the names assigned to the proof templates, the reason is the following:

1. If the proof template is tagged with X-intro, the connective X appears in the conclusion of the proof template; it is introduced. For example, in Proof Template 1.2, the conclusion is $P \Rightarrow Q$, and $\Rightarrow$ is indeed introduced.

2. If the proof template is tagged with X-Elim, the connective X appears in one of the premises of the proof template but it does not appear in the conclusion; it is eliminated. For example, in Proof Template 1.3 (modus ponens), $P \Rightarrow Q$ occurs as a premise but the conclusion is Q; the symbol $\Rightarrow$ has been eliminated.

The introduction/elimination pattern is a characteristic of the kind of proof system that we are describing which is called a *natural deduction proof system*.

Example 1.1. Let us give a simple example of the use of Proof Template 1.2. Recall that a natural number n is odd iff it is of the form $2k + 1$, where $k \in \mathbb{N}$. Let us denote the fact that a number n is odd by $\text{odd}(n)$. We would like to prove the implication

$$\text{odd}(n) \Rightarrow \text{odd}(n + 2).$$

Following Proof Template 1.2, we add $\text{odd}(n)$ as a premise (which means that we take as proven the fact that n is odd) and we try to conclude that $n + 2$ must be odd. However, to say that n is odd is to say that $n = 2k + 1$ for some natural number k. Now,

$$n + 2 = 2k + 1 + 2 = 2(k + 1) + 1,$$

which means that $n + 2$ is odd. (Here, $n = 2h + 1$, with $h = k + 1$, and $k + 1$ is a natural number because k is.)

Thus, we proven that *if we assume* odd(n)*, then we can conclude* odd$(n+2)$, and according to Proof Template 1.2, by step (3) we delete the premise odd(n) and we obtain a proof of the proposition

$$\text{odd}(n) \Rightarrow \text{odd}(n+2).$$

It should be noted that the above proof of the proposition odd$(n) \Rightarrow$ odd$(n+2)$ *does not depend* on any premises (other than the implicit fact that we are assuming n is a natural number). In particular, this proof does not depend on the premise odd(n), which was assumed (became "active") during our subproof step. Thus, after having applied the Proof Template 1.2, we made sure that the premise odd(n) is deactivated.

Example 1.2. For a second example, we wish to prove the proposition $P \Rightarrow P$.

According to Proof Template 1.2, we assume P. But then, by Proof Template 1.1, we obtain a deduction with premise P and conclusion P; by executing step (3) of Proof Template 1.2, the premise P is deleted, and we obtain a deduction of $P \Rightarrow P$ from the empty list of premises. Thank God, $P \Rightarrow P$ is provable!

Proofs described in words as above are usually better understood when represented as trees. We will reformulate our proof templates in tree form and explain very precisely how to build proofs as trees in Chapter 2. For now, we use tree representations of proofs in an informal way.

1.4 Proof Trees and Deduction Trees

A proof tree is drawn with its leaves at the top, corresponding to assumptions, and its root at the bottom, corresponding to the conclusion. In computer science, trees are usually drawn with their root at the top and their leaves at the bottom, but proof trees are drawn as the trees that we see in nature. Instead of linking nodes by edges, it is customary to use horizontal bars corresponding to the proof templates. One or more nodes appear as premises above a vertical bar, and the conclusion of the proof template appears immediately below the vertical bar.

According to the first step of proof of $P \Rightarrow P$ (presented in words) we move the premise P to the list of premises, building a deduction of the conclusion P from the premise P corresponding to the following unfinished tree in which some leaf is labeled with the premise P but with a missing subtree establishing P as the conclusion:

$$\dfrac{\begin{array}{c} P^x \\ P \end{array}}{P \Rightarrow P} \quad \text{Implication-Intro } x$$

The premise P is tagged with the label x which corresponds to the proof rule which causes its deletion from the list of premises.

In order to obtain a proof we need to apply a proof template which allows use to deduce P from P and of course this is the Trivial Deduction proof template.

The finished proof is represented by the tree shown below. Observe that the premise P is tagged with the symbol $\surd$, which means that it has been deleted from the list of premises. The tree representation of proofs also has the advantage that we can tag the premises in such a way that each tag indicates which rule causes the corresponding premise to be deleted. In the tree below, the premise P is tagged with x, and it is deleted when the proof template indicated by x is applied.

$$\dfrac{\dfrac{P^{x\surd}}{P}\ \text{Trivial Deduction}}{P \Rightarrow P}\ \text{Implication-Intro}\ x$$

Example 1.3. For a third example, we prove the proposition $P \Rightarrow (Q \Rightarrow P)$.

According to Proof Template 1.2, we assume P as a premise and we try to prove $Q \Rightarrow P$ assuming P. In order to prove $Q \Rightarrow P$, by Proof Template 1.2, we assume Q as a new premise so the set of premises becomes $\{P, Q\}$, and then we try to prove P from P and Q.

At this stage we have the following unfinished tree with two leaves labeled P and Q but with a missing subtree establishing P as the conclusion:

$$\dfrac{\dfrac{\begin{matrix} P^x, Q^y \\ P \end{matrix}}{Q \Rightarrow P}\ \text{Implication-Intro}\ y}{P \Rightarrow (Q \Rightarrow P)}\ \text{Implication-Intro}\ x$$

We need to find a deduction of P from the premises P and Q. By Proof Template 1.1 (trivial deductions), we have a deduction with the list of premises $\{P, Q\}$ and conclusion P. Then, executing step (3) of Proof Template 1.2 twice, we delete the premises Q, and then the premise P (in this order), and we obtain a proof of $P \Rightarrow (Q \Rightarrow P)$. The above proof of $P \Rightarrow (Q \Rightarrow P)$ (presented in words) is represented by the following tree:

$$\dfrac{\dfrac{\dfrac{P^{x\surd}, Q^{y\surd}}{P}\ \text{Trivial Deduction}}{Q \Rightarrow P}\ \text{Implication-Intro}\ y}{P \Rightarrow (Q \Rightarrow P)}\ \text{Implication-Intro}\ x$$

Observe that both premises P and Q are tagged with the symbol $\surd$, which means that they have been deleted from the list of premises.

We tagged the premises in such a way that each tag indicates which rule causes the corresponding premise to be deleted. In the above tree, Q is tagged with y, and it is deleted when the proof template indicated by y is applied, and P is tagged with x, and it is deleted when the proof template indicated by x is applied. In a proof, all leaves must be tagged with the symbol $\surd$.

Example 1.4. Let us now give a proof of $P \Rightarrow ((P \Rightarrow Q) \Rightarrow Q)$.

Using Proof Template 1.2, we assume both P and $P \Rightarrow Q$ and we try to prove Q. At this stage we have the following unfinished tree with two leaves labeled $P \Rightarrow Q$ and P but with a missing subtree establishing Q as the conclusion:

$$\dfrac{\dfrac{\begin{matrix}(P \Rightarrow Q)^x & & P^y\\ & Q & \end{matrix}}{(P \Rightarrow Q) \Rightarrow Q}\ \text{Implication-Intro}\ x}{P \Rightarrow ((P \Rightarrow Q) \Rightarrow Q)}\ \text{Implication-Intro}\ y$$

We can use Proof Template 1.3 to derive a deduction of Q from $P \Rightarrow Q$ and P. Finally, we execute step (3) of Proof Template 1.2 to delete $P \Rightarrow Q$ and P (in this order), and we obtain a proof of $P \Rightarrow ((P \Rightarrow Q) \Rightarrow Q)$. A tree representation of the above proof is shown below.

$$\dfrac{\dfrac{\dfrac{(P \Rightarrow Q)^{x\surd} \qquad P^{y\surd}}{Q}\ \text{Implication-Elim}}{(P \Rightarrow Q) \Rightarrow Q}\ \text{Implication-Intro}\ x}{P \Rightarrow ((P \Rightarrow Q) \Rightarrow Q)}\ \text{Implication-Intro}\ y$$

Remark: We have not yet examined how we how we can represent precisely arbitrary deductions. This can be done using certain types of trees where the nodes are tagged with lists of premises. Two methods for doing this are carefully defined in Chapter 2. It turns out that the same premise may be used in more than one location in the tree, but in our informal presentation, we ignore such fine details.

We now describe the proof templates dealing with the connectives $\neg, \wedge, \vee, \equiv$.

1.5 Proof Templates for $\neg$

Recall that $\neg P$ is an abbreviation for $P \Rightarrow \perp$. We begin with the proof templates for negation, for direct proofs.

Proof Template 1.4. *(Negation–Intro)*

Given a list Γ of premises (possibly empty), to obtain a deduction with conclusion $\neg P$, proceed as follows:

1. *Add one or more occurrences of P as additional premises to the list Γ.*
2. *Derive a contradiction. More precisely, make a deduction of the conclusion $\perp$ from P and the premises in Γ.*

3. *Delete P from the list of premises.*

Proof Template 1.4 is a special case of Proof Template 1.2, since $\neg P$ is an abbreviation for $P \Rightarrow \perp$.

Proof Template 1.5. *(Negation–Elim)*

Given a deduction with conclusion $\neg P$ from a list of premises Γ and a deduction with conclusion P from a list of premises Δ, we obtain a contradiction; that is, a deduction with conclusion $\perp$. The list of premises of this new deduction is Γ, Δ.

Proof Template 1.5 is a special case of Proof Template 1.3, since $\neg P$ is an abbreviation for $P \Rightarrow \perp$.

Proof Template 1.6. *(Perp–Elim)*

Given a deduction with conclusion $\perp$ (a contradiction), for every proposition Q, we obtain a deduction with conclusion Q. The list of premises of this new deduction is the same as the original list of premises.

The last proof template for negation constructs an indirect proof; it is the *proof–by–contradiction* principle.

Proof Template 1.7. *(Proof–By–Contradiction Principle)*

Given a list Γ of premises (possibly empty), to obtain a deduction with conclusion P, proceed as follows:

1. *Add one of more occurrences of $\neg P$ as additional premises to the list Γ.*

2. *Derive a contradiction. More precisely, make a deduction of the conclusion $\perp$ from $\neg P$ and the premises in Γ.*

3. *Delete $\neg P$ from the list of premises.*

Proof Template 1.7 (the proof–by–contradiction principle) also has the fancy name of *reductio ad absurdum rule*, for short *RAA*.

Proof Template 1.6 may seem silly and one might wonder why we stated it. It turns out that it is subsumed by Proof Template 1.7, but it is still useful to state it as a proof template.

Example 1.5. Let us prove that for every natural number n, if n^2 is odd, then n itself must be odd.

We use the proof–by–contradiction principle (Proof Template 1.7), so we assume that n is not odd, which means that n is even. (Actually, in this step we are using a property of the natural numbers that is proven by induction but let's not worry about that right now; a proof can be found in Section 1.12) But to say that n is even means that $n = 2k$ for some k and then $n^2 = 4k^2 = 2(2k^2)$, so n^2 is even, contradicting the assumption that n^2 is odd. By the proof–by–contradiction principle (Proof Template 1.7), we conclude that n must be odd.

Example 1.6. Let us prove that $\neg\neg P \Rightarrow P$.

It turns out that this requires using the proof–by–contradiction principle (Proof Template 1.7). First by Proof Template 1.2, assume $\neg\neg P$ as a premise. Then by the proof–by–contradiction principle (Proof template 1.7), in order to prove P, assume $\neg P$. By Proof Template 1.5, we obtain a contradiction ($\perp$). Thus, by step (3) of the proof–by–contradiction principle (Proof Template 1.7), we delete the premise $\neg P$ and we obtain a deduction of P from $\neg\neg P$. Finally, by step (3) of Proof Template 1.2, we delete the premise $\neg\neg P$ and obtain a proof of $\neg\neg P \Rightarrow P$. This proof has the following tree representation.

$$\dfrac{\dfrac{\dfrac{\neg\neg P^{y\surd} \qquad \neg P^{x\surd}}{\perp}\ \text{Negation-Elim}}{P}\ \text{RAA}\ x}{\neg\neg P \Rightarrow P}\ \text{Implication-Intro}\ y$$

Example 1.7. Now, we prove that $P \Rightarrow \neg\neg P$.

First by Proof Template 1.2, assume P as a premise. In order to prove $\neg\neg P$ from P, by Proof Template 1.4, assume $\neg P$. We now have the two premises $\neg P$ and P, so by Proof Template 1.5, we obtain a contradiction ($\perp$). By step (3) of Proof Template 1.4, we delete the premise $\neg P$ and we obtain a deduction of $\neg\neg P$ from P. Finally, by step (3) of Proof Template 1.2, delete the premise P to obtain a proof of $P \Rightarrow \neg\neg P$. This proof has the following tree representation.

$$\dfrac{\dfrac{\dfrac{\neg P^{x\surd} \qquad P^{y\surd}}{\perp}\ \text{Negation-Elim}}{\neg\neg P}\ \text{Negation-Intro}\ x}{P \Rightarrow \neg\neg P}\ \text{Implication-Intro}\ y$$

Observe that the previous two examples show that the equivalence $P \equiv \neg\neg P$ is provable. As a consequence of this equivalence, if we prove a negated proposition $\neg P$ using the proof–by–contradiction principle, we assume $\neg\neg P$ and we deduce a contradiction. But since $\neg\neg P$ and P are equivalent (as far as provability), this amounts to deriving a contradiction from P, which is just the Proof Template 1.4.

In summary, *to prove a negated proposition $\neg P$, always use Proof Template 1.4.*

On the other hand, to prove a nonnegated proposition, it is generally not possible to tell if a direct proof exists or if the proof–by–contradiction principle is required. There are propositions for which it is required, for example $\neg\neg P \Rightarrow P$ and $(\neg(P \Rightarrow Q)) \Rightarrow P$.

Example 1.8. Let us now prove that $(\neg(P \Rightarrow Q)) \Rightarrow \neg Q$.

First by Proof Template 1.2, we add $\neg(P \Rightarrow Q)$ as a premise. Then, in order to prove $\neg Q$ from $\neg(P \Rightarrow Q)$, we use Proof Template 1.4 and we add Q as a premise. Now, recall that we showed in Example 1.3 that $P \Rightarrow Q$ is provable assuming Q (with P and Q switched).

Then since $\neg(P \Rightarrow Q)$ is a premise, by Proof Template 1.5, we obtain a deduction of $\perp$. We now execute step (3) of Proof Template 1.4, delete the premise Q to obtain a deduction of $\neg Q$ from $\neg(P \Rightarrow Q)$, we and we execute step (3) of Proof Template 1.2 to delete the premise $\neg(P \Rightarrow Q)$ and obtain a proof of $(\neg(P \Rightarrow Q)) \Rightarrow \neg Q$. The above proof corresponds to the following tree.

$$
\dfrac{\dfrac{\dfrac{\neg(P \Rightarrow Q)^{z\surd} \qquad \dfrac{\dfrac{Q^{y\surd} \qquad P^{x\surd}}{Q}\ \text{Trivial Deduction}}{P \Rightarrow Q}\ \text{Implication-Intro}\ x}{\perp}\ \text{Negation-Elim}}{\neg Q}\ \text{Negation-Intro}\ y}{(\neg(P \Rightarrow Q)) \Rightarrow \neg Q}\ \text{Implication-Intro}\ z
$$

Here is an example using Proof Templates 1.6 (Perp–Elim) and 1.7 (RAA).

Example 1.9. Let us prove that $(\neg(P \Rightarrow Q)) \Rightarrow P$.

First we use Proof Template 1.2, and we assume $\neg(P \Rightarrow Q)$ as a premise. Next we use the proof–by–contradiction principle (Proof Template 1.7). So, in order to prove P, we assume $\neg P$ as another premise. The next step is to deduce $P \Rightarrow Q$. By Proof Template 1.2, we assume P as an additional premise. By Proof Template 1.5, from $\neg P$ and P we obtain a deduction of $\perp$, and then by Proof Template 1.6 a deduction of Q from $\neg P$ and P. By Proof Template 1.2, executing step (3), we delete the premise P and we obtain a deduction of $P \Rightarrow Q$. At this stage, we have the premises $\neg P, \neg(P \Rightarrow Q)$ and a deduction of $P \Rightarrow Q$, so by Proof Template 1.5, we obtain a deduction of $\perp$. This is a contradiction, so by step (3) of the proof–by–contradiction principle (Proof Template 1.7) we can delete the premise $\neg P$, and we have a deduction of P from $\neg(P \Rightarrow Q)$. Finally, we execute step (3) of Proof Template 1.2 and delete the premise $\neg(P \Rightarrow Q)$, which yields the desired proof of $(\neg(P \Rightarrow Q)) \Rightarrow P$. The above proof has the following tree representation.

$$
\dfrac{\dfrac{\dfrac{\neg(P \Rightarrow Q)^{z\surd} \qquad \dfrac{\dfrac{\dfrac{\neg P^{y\surd} \qquad P^{x\surd}}{\perp}\ \text{Negation-Elim}}{Q}\ \text{Perp-Elim}}{P \Rightarrow Q}\ \text{Implication-Intro}\ x}{\perp}\ \text{Negation-Elim}}{P}\ \text{RAA}\ y}{(\neg(P \Rightarrow Q)) \Rightarrow P}\ \text{Implication-Intro}\ z
$$

The reader may be surprised by how many steps are needed in the above proof and may wonder whether the proof–by–contradiction principle is actually needed. It can be shown that the proof–by–contradiction principle must be used, and unfortuately there is no shorter proof.

Even though Proof Template 1.4 qualifies as a direct proof template, it proceeds by deriving a contradiction, so I suggest to call it the *proof–by–contradiction for negated propositions* principle.

Remark: The fact that the implication $\neg\neg P \Rightarrow P$ is provable has the interesting consequence that if we take $\neg\neg P \Rightarrow P$ as an *axiom* (which means that $\neg\neg P \Rightarrow P$ is assumed to be provable without requiring any proof), then the proof–by–contradiction principle (Proof Template 1.7) becomes redundant. Indeed, Proof Template 1.7 is subsumed by Proof Template 1.4, because if we have a deduction of $\perp$ from $\neg P$, then by Proof Template 1.4 we delete the premise ¬P to obtain a deduction of $\neg\neg P$. Since $\neg\neg P \Rightarrow P$ is assumed to be provable, by Proof Template 1.3, we get a proof of P. The tree shown below illustrates what is going on. In this tree, a proof of $\perp$ from the premise $\neg P$ is denoted by $\mathcal{D}$.

$$\dfrac{\neg\neg P \Rightarrow P \qquad \dfrac{\begin{matrix}\neg P^{x}\checkmark \\ \mathcal{D} \\ \perp\end{matrix}}{\neg\neg P}\ \text{Negation-Intro}\ x}{P}\ \text{Implication-Elim}$$

Proof Templates 1.5 and 1.6 together imply that if a contradiction is obtained during a deduction because two inconsistent propositions P and $\neg P$ are obtained, then *all* propositions are provable (anything goes). This explains why mathematicians are leary of inconsistencies.

1.6 Proof Templates for $\wedge, \vee, \equiv$

The proof templates for conjunction are the simplest.

Proof Template 1.8. *(And–Intro)*

Given a deduction with conclusion P from a list of premises Γ and a deduction with conclusion Q from a list of premises Δ, we obtain a deduction with conclusion $P \wedge Q$. The list of premises of this new deduction is Γ, Δ.

Proof Template 1.9. *(And–Elim)*

Given a deduction with conclusion $P \wedge Q$, we obtain a deduction with conclusion P, and a deduction with conclusion Q. The list of premises of these new deductions is the same as the list of premises of the orginal deduction.

Let us consider a few examples of proofs using the proof templates for conjunction as well as Proof Templates 1.4 and 1.7.

Example 1.10. Let us prove that for any natural number n, if n is divisible by 2 and n is divisible by 3, then n is divisible by 6. This is expressed by the proposition

$$((2 \mid n) \wedge (3 \mid n)) \Rightarrow (6 \mid n).$$

We start by using Proof Templates 1.2 and we add the premise $(2 \mid n) \wedge (3 \mid n)$. Using Proof Template 1.9 twice, we obtain deductions of $(2 \mid n)$ and $(3 \mid n)$ from $(2 \mid n) \wedge (3 \mid n)$. But $(2 \mid n)$ means that

$$n = 2a$$

for some $a \in \mathbb{N}$, and $3 \mid n$ means that

$$n = 3b$$

for some $b \in \mathbb{N}$. This implies that

$$n = 2a = 3b.$$

Because 2 and 3 are relatively prime (their only common divisor is 1), the number 2 must divide b (and 3 must divide a) so $b = 2c$ for some $c \in \mathbb{N}$. Here we are using Euclid's lemma. So, we have shown that

$$n = 3b = 3 \cdot 2c = 6c,$$

which says that n is divisible by 6. We conclude with step (3) of Proof Template 1.2 by deleting the premise $(2 \mid n) \wedge (3 \mid n)$ and we obtain our proof.

Example 1.11. Let us prove that for any natural number n, if n is divisible by 6, then n is divisible by 2 and n is divisible by 3. This is expressed by the proposition

$$(6 \mid n) \Rightarrow ((2 \mid n) \wedge (3 \mid n)).$$

We start by using Proof Template 1.2 and we add the premise $6 \mid n$. This means that

$$n = 6a = 2 \cdot 3a$$

for some $a \in \mathbb{N}$. This implies that $2 \mid n$ and $3 \mid n$, so we have a deduction of $2 \mid n$ from the premise $6 \mid n$ and a deduction of $3 \mid n$ from the premise $6 \mid n$. By Proof Template 1.8, we obtain a deduction of $(2 \mid n) \wedge (3 \mid n)$ from $6 \mid n$, and we apply step (3) of Proof Template 1.2 to delete the premise $6 \mid n$ and obtain our proof.

Example 1.12. Let us prove that a natural number n cannot be even and odd simultaneously. This is expressed as the proposition

$$\neg(\mathrm{odd}(n) \wedge \mathrm{even}(n)).$$

We begin with Proof Template 1.4 and we assume $\mathrm{odd}(n) \wedge \mathrm{even}(n)$ as a premise. Using Proof Template 1.9 twice, we obtain deductions of $\mathrm{odd}(n)$ and $\mathrm{even}(n)$ from $\mathrm{odd}(n) \wedge \mathrm{even}(n)$. Now $\mathrm{odd}(n)$ says that $n = 2a + 1$ for some $a \in \mathbb{N}$, and $\mathrm{even}(n)$ says that $n = 2b$ for some $b \in \mathbb{N}$. But then,

$$n = 2a + 1 = 2b,$$

so we obtain $2(b-a)=1$. Since $b-a$ is an integer, either $2(b-a)=0$ (if $a=b$) or $|2(b-a)| \geq 2$, so we obtain a contradiction. Applying step (3) of Proof Template 1.4, we delete the premise $\mathrm{odd}(n) \wedge \mathrm{even}(n)$ and we have a proof of $\neg(\mathrm{odd}(n) \wedge \mathrm{even}(n))$.

Example 1.13. Let us prove that $(\neg(P \Rightarrow Q)) \Rightarrow (P \wedge \neg Q)$.

We start by using Proof Templates 1.2 and we add $\neg(P \Rightarrow Q)$ as a premise. Now, in Example 1.9 we showed that $(\neg(P \Rightarrow Q)) \Rightarrow P$ is provable, and this proof contains a deduction of P from $\neg(P \Rightarrow Q)$. Similarly, in Example 1.8 we showed that $(\neg(P \Rightarrow Q)) \Rightarrow \neg Q$ is provable, and this proof contains a deduction of $\neg Q$ from $\neg(P \Rightarrow Q)$. By proof Template 1.8, we obtain a deduction of $P \wedge \neg Q$ from $\neg(P \Rightarrow Q)$, and executing step (3) of Proof Templates 1.2, we obtain a proof of $(\neg(P \Rightarrow Q)) \Rightarrow (P \wedge \neg Q)$. The following tree represents the above proof. Observe that *two copies* of the premise $\neg(P \Rightarrow Q)$ are needed.

$$\dfrac{\dfrac{\dfrac{\neg(P \Rightarrow Q)^{z}\surd \qquad \dfrac{\dfrac{\dfrac{\neg P^{y}\surd \qquad P^{x}\surd}{\perp}}{Q}}{P \Rightarrow Q}\, x}{\dfrac{\perp}{P}}\ \text{RAA}\ y \qquad \dfrac{\neg(P \Rightarrow Q)^{z}\surd \qquad \dfrac{\dfrac{Q^{w}\surd \qquad P^{t}\surd}{Q}}{P \Rightarrow Q}\, t}{\dfrac{\perp}{\neg Q}}\ \text{Negation-Intro}\ w}{P \wedge \neg Q}}{(\neg(P \Rightarrow Q)) \Rightarrow (P \wedge \neg Q)}\, z$$

Next, we present the proof templates for disjunction.

Proof Template 1.10. *(Or–Intro)*

Given a list Γ *of premises (possibly empty),*

1. *If we have a deduction with conclusion* P*, then we obtain a deduction with conclusion* $P \vee Q$.
2. *If we have a deduction with conclusion* Q*, then we obtain a deduction with conclusion* $P \vee Q$.

In both cases, the new deduction has Γ *as premises.*

Proof Template 1.11. *(Or–Elim or Proof–By–Cases)*

Given three lists of premises Γ, Δ, Λ, *to obtain a deduction of some proposition* R *as conclusion, proceed as follows:*

1. *Construct a deduction of some disjunction* $P \vee Q$ *from the list of premises* Γ.
2. *Add one or more occurrences of* P *as additional premises to the list* Δ *and find a deduction of* R *from* P *and* Δ.

3. *Add one or more occurrences of Q as additional premises to the list Λ and find a deduction of R from Q and Λ.*

The list of premises after applying this rule is Γ, Δ, Λ.

Note that in making the two deductions of R, the premise $P \vee Q$ is *not* assumed.

Example 1.14. Let us show that for any natural number n, if 4 divides n or 6 divides n, then 2 divides n. This can expressed as

$$((4 \mid n) \vee (6 \mid n)) \Rightarrow (2 \mid n).$$

First, by Proof Template 1.2, we assume $(4 \mid n) \vee (6 \mid n)$ as a premise. Next, we use Proof Template 1.11, the proof–by–cases principle. First, assume $(4 \mid n)$. This means that

$$n = 4a = 2 \cdot 2a$$

for some $a \in \mathbb{N}$. Therefore, we conclude that $2 \mid n$. Next, assume $(6 \mid n)$. This means that

$$n = 6b = 2 \cdot 3b$$

for some $b \in \mathbb{N}$. Again, we conclude that $2 \mid n$. Since $(4 \mid n) \vee (6 \mid n)$ is a premise, by Proof Template 1.11, we can obtain a deduction of $2 \mid n$ from $(4 \mid n) \vee (6 \mid n)$. Finally, by Proof Template 1.2, we delete the premise $(4 \mid n) \vee (6 \mid n)$ to obtain our proof.

Proof Template 1.10 (Or–Intro) may seem trivial, so let us show an example illustrating its use.

Example 1.15. Let us prove that $\neg(P \vee Q) \Rightarrow (\neg P \wedge \neg Q)$.

First by Proof Template 1.2, we assume $\neg(P \vee Q)$ (two copies). In order to derive $\neg P$, by Proof Template 1.4, we also assume P. Then by Proof Template 1.10 we deduce $P \vee Q$, and since we have the premise $\neg(P \vee Q)$, by Proof Template 1.5 we obtain a contradiction. By Proof Template 1.4, we can delete the premise P and obtain a deduction of $\neg P$ from $\neg(P \vee Q)$.

In a similar way we can construct a deduction of $\neg Q$ from $\neg(P \vee Q)$. By Proof Template 1.8, we get a deduction of $\neg P \wedge \neg Q$ from $\neg(P \vee Q)$, and we finish by applying Proof Template 1.2. A tree representing the above proof is shown below.

$$
\dfrac{\dfrac{\neg(P \vee Q)^{z\checkmark} \qquad \dfrac{P^{x\checkmark}}{P \vee Q}\ \text{Or-Intro}}{\dfrac{\bot}{\neg P}\ \text{Negation-Intro } x} \qquad \dfrac{\neg(P \vee Q)^{z\checkmark} \qquad \dfrac{Q^{w\checkmark}}{P \vee Q}\ \text{Or-Intro}}{\dfrac{\bot}{\neg Q}\ \text{Negation-Intro } w}}{\dfrac{\neg P \wedge \neg Q}{\neg(P \vee Q) \Rightarrow (\neg P \wedge \neg Q)}\ z}
$$

The proposition $(\neg P \land \neg Q) \Rightarrow \neg(P \lor Q)$ is also provable using the proof–by–cases principle. Here is a proof tree; we leave it as an exercise to the reader to check that the proof templates have been applied correctly.

$$\cfrac{\cfrac{\cfrac{(P \lor Q)^{z\surd} \qquad \cfrac{\cfrac{(\neg P \land \neg Q)^{t\surd}}{\neg P} \qquad P^{x\surd}}{\bot} \qquad \cfrac{\cfrac{(\neg P \land \neg Q)^{t\surd}}{\neg Q} \qquad Q^{y\surd}}{\bot}}{\bot}\;{\scriptstyle x,y}}{\neg(P \lor Q)}\;{\scriptstyle z}}{(\neg P \land \neg Q) \Rightarrow \neg(P \lor Q)}\;{\scriptstyle t}$$

As a consequence the equivalence

$$\neg(P \lor Q) \equiv (\neg P \land \neg Q)$$

is provable. This is one of three identities known as *de Morgan laws*.

Example 1.16. Next let us prove that $\neg(\neg P \lor \neg Q) \Rightarrow P$.

First by Proof Template 1.2, we assume $\neg(\neg P \lor \neg Q)$ as a premise. In order to prove P from $\neg(\neg P \lor \neg Q)$, we use the proof–by–contradiction principle (Proof Template 1.7). So, we add $\neg P$ as a premise. Now, by Proof Template 1.10, we can deduce $\neg P \lor \neg Q$ from $\neg P$, and since $\neg(\neg P \lor \neg Q)$ is a premise, by Proof Template 1.5, we obtain a contradiction. By the proof–by–contradiction principle (Proof Template 1.7), we delete the premise $\neg P$ and we obtain a deduction of P from $\neg(\neg P \lor \neg Q)$. We conclude by using Proof Template 1.2 to delete the premise $\neg(\neg P \lor \neg Q)$ and to obtain our proof. A tree representing the above proof is shown below.

$$\cfrac{\cfrac{\cfrac{\neg(\neg P \lor \neg Q)^{y\surd} \qquad \cfrac{\neg P^{x\surd}}{\neg P \lor \neg Q}}{\bot}}{P}\;\text{RAA}\ {\scriptstyle x}}{\neg(\neg P \lor \neg Q) \Rightarrow P}\;{\scriptstyle y}$$

A similar proof shows that $\neg(\neg P \lor \neg Q) \Rightarrow Q$ is provable. Putting together the proofs of P and Q from $\neg(\neg P \lor \neg Q)$ using Proof Template 1.8, we obtain a proof of

$$\neg(\neg P \lor \neg Q) \Rightarrow (P \land Q).$$

A tree representing this proof is shown below.

$$\dfrac{\dfrac{\dfrac{\neg(\neg P \lor \neg Q)^{y}\checkmark \qquad \dfrac{\neg P^{x}\checkmark}{\neg P \lor \neg Q}}{\dfrac{\perp}{P}\ \text{RAA}\ x} \qquad \dfrac{\neg(\neg P \lor \neg Q)^{y}\checkmark \qquad \dfrac{\neg Q^{w}\checkmark}{\neg P \lor \neg Q}}{\dfrac{\perp}{Q}\ \text{RAA}\ w}}{P \land Q}}{\neg(\neg P \lor \neg Q) \Rightarrow (P \land Q)}\ y$$

Example 1.17. The proposition $\neg(P \land Q) \Rightarrow (\neg P \lor \neg Q)$ is provable.

First by Proof Template 1.2, we assume $\neg(P \land Q)$ as a premise. Next we use the proof–by–contradiction principle (Proof Template 1.7) to deduce $\neg P \lor \neg Q$, so we also assume $\neg(\neg P \lor \neg Q)$. Now, we just showed that $P \land Q$ is provable from the premise $\neg(\neg P \lor \neg Q)$. Using the premise $\neg(P \land Q)$, by Proof Principle 1.5, we derive a contradiction, and by the proof–by–contradiction principle, we delete the premise $\neg(\neg P \lor \neg Q)$ to obtain a deduction of $\neg P \lor \neg Q$ from $\neg(P \land Q)$. We finish the proof by applying Proof Template 1.2. This proof is represented by the following tree.

$$\dfrac{\dfrac{\neg(P \land Q)^{t}\checkmark \qquad \dfrac{\dfrac{\neg(\neg P \lor \neg Q)^{y}\checkmark \qquad \dfrac{\neg P^{x}\checkmark}{\neg P \lor \neg Q}}{\dfrac{\perp}{P}\ \text{RAA}\ x} \qquad \dfrac{\neg(\neg P \lor \neg Q)^{y}\checkmark \quad \dfrac{\neg Q^{w}\checkmark}{\neg P \lor \neg Q}}{\dfrac{\perp}{Q}\ \text{RAA}\ w}}{P \land Q}}{\dfrac{\perp}{\neg P \lor \neg Q}\ \text{RAA}\ y}}{\neg(P \land Q) \Rightarrow \neg P \lor \neg Q}\ t$$

The next example is particularly interesting.

It can be shown that the proof–by–contradiction principle must be used.

Example 1.18. We prove the proposition

$$P \lor \neg P.$$

We use the proof–by–contradiction principle (Proof Template 1.7), so we assume $\neg(P \lor \neg P)$ as a premise. The first tricky part of the proof is that we actually assume that we have two copies of the premise $\neg(P \lor \neg P)$.

Next the second tricky part of the proof is that using one of the two copies of $\neg(P \lor \neg P)$, we are going to deduce $P \lor \neg P$. For this, we first derive $\neg P$ using Proof Template 1.4, so we assume P. By Proof Template 1.10, we deduce $P \lor \neg P$, but we have the premise $\neg(P \lor \neg P)$, so by Proof Template 1.5, we obtain a contradiction. Next, by Proof Template 1.4 we delete the premise P, deduce $\neg P$, and then by Proof Template 1.10 we deduce $P \lor \neg P$.

Since we still have a second copy of the premise $\neg(P \vee \neg P)$, by Proof Template 1.5, we get a contradiction! The only premise left is $\neg(P \vee \neg P)$ (two copies of it), so by the proof–by–contradiction principle (Proof Template 1.7), we delete the premise $\neg(P \vee \neg P)$ and we obtain the desired proof of $P \vee \neg P$.

$$\dfrac{\dfrac{\neg(P \vee \neg P)^{x}\surd \qquad \dfrac{\dfrac{\dfrac{\neg(P \vee \neg P)^{x}\surd \qquad \dfrac{P^{y}\surd}{P \vee \neg P}}{\perp}\ \text{Negation-Elim}}{\neg P}\ \text{Negation-Intro}\ y}{P \vee \neg P}}{\perp}\ \text{Negation-Elim}}{P \vee \neg P}\ \text{RAA}\ x$$

If the above proof made you dizzy, this is normal. The sneaky part of this proof is that when we proceed by contradiction and assume $\neg(P \vee \neg P)$, this proposition is an inconsistency, so it allows us to derive $P \vee \neg P$, which then clashes with $\neg(P \vee \neg P)$ to yield a contradiction. Observe that during the proof we actually showed that $\neg\neg(P \vee \neg P)$ is provable. The proof–by–contradiction principle is needed to get rid of the double negation.

The fact is that even though the proposition $P \vee \neg P$ seems obviously "true," its truth is viewed as controversial by certain matematicians and logicians. To some extant, this is why its proof has to be a bit tricky and has to involve the proof–by–contradiction principle. This matter is discussed quite extensively in Chapter 2. In this chapter, which is more informal, let us simply say that the proposition $P \vee \neg P$ is known as the *law of excluded middle*. Indeed, intuitively, it says that for every proposition P, either P is true or ¬P is true; there is no middle alternative.

It can be shown that if we take all formulae of the form $P \vee \neg P$ as axioms, then the proof–by–contradiction principle is derivable from the other proof tempates; see Section 2.7. Furthermore, the proposition $\neg\neg P \Rightarrow P$ and $P \vee \neg P$ are equivalent (that is, $(\neg\neg P \Rightarrow P) \equiv (P \vee \neg P)$ is provable).

Typically, to prove a disjunction $P \vee Q$, it is rare that we can use Proof Template 1.10 (Or–Intro), because this requires constructing of a proof of P or a proof of Q in the first place. But the fact that $P \vee Q$ is provable does not imply in general that either a proof of P or a proof of Q can be produced, as the example of the proposition $P \vee \neg P$ shows (other examples can be given). Thus, *usually to prove a disjunction we use the proof–by-contradiction principle*. Here is an example.

Example 1.19. Given some natural numbers p, q, we wish to prove that if 2 divides pq, then either 2 divides p or 2 divides q. This can be expressed by

$$(2 \mid pq) \Rightarrow ((2 \mid p) \vee (2 \mid q)).$$

We use the proof–by-contradiction principle (Proof Template 1.7), so we assume $\neg((2 \mid p) \vee (2 \mid q))$ as a premise. This is a proposition of the form $\neg(P \vee Q)$, and in Example 1.15

we showed that $\neg(P \vee Q) \Rightarrow (\neg P \wedge \neg Q)$ is provable. Thus, by Proof Template 1.3, we deduce that $\neg(2 \mid p) \wedge \neg(2 \mid q)$. By Proof Template 1.9, we deduce both $\neg(2 \mid p)$ and $\neg(2 \mid q)$. Using some basic arithmetic, this means that $p = 2a + 1$ and $q = 2b + 1$ for some $a, b \in \mathbb{N}$. But then,

$$pq = 2(2ab + a + b) + 1.$$

and pq is not divisible by 2, a contradiction. By the proof–by-contradiction principle (Proof Template 1.7), we can delete the premise $\neg((2 \mid p) \vee (2 \mid q))$ and obtain the desired proof.

Another proof template which is convenient to use in some cases is the *proof–by–contrapositive principle*.

Proof Template 1.12. *(Proof–By–Contrapositive)*

Given a list of premises Γ, to prove an implication $P \Rightarrow Q$, proceed as follows:

1. *Add $\neg Q$ to the list of premises Γ.*
2. *Construct a deduction of $\neg P$ from the premises $\neg Q$ and Γ.*
3. *Delete $\neg Q$ from the list of premises.*

It is not hard to see that the proof–by–contrapositive principle (Proof Template 1.12) can be derived from the proof–by–contradiction principle. We leave this as an exercise.

Example 1.20. We prove that for any two natural numbers $m, n \in \mathbb{N}$, if $m + n$ is even, then m and n have the same parity. This can be expressed as

$$\text{even}(m + n) \Rightarrow ((\text{even}(m) \wedge \text{even}(n)) \vee (\text{odd}(m) \wedge \text{odd}(n))).$$

According to Proof Template 1.12 (proof–by–contrapositive principle), let us assume $\neg((\text{even}(m) \wedge \text{even}(n)) \vee (\text{odd}(m) \wedge \text{odd}(n)))$. Using the implication proven in Example 1.15 $((\neg(P \vee Q)) \Rightarrow \neg P \wedge \neg Q))$ and Proof Template 1.3, we deduce that $\neg(\text{even}(m) \wedge \text{even}(n))$ and $\neg(\text{odd}(m) \wedge \text{odd}(n))$. Using the result of Example 1.17 and modus ponens (Proof Template 1.3), we deduce that $\neg\text{even}(m) \vee \neg\text{even}(n)$ and $\neg\text{odd}(m) \vee \neg\text{odd}(n)$. At this point, we can use the proof–by–cases principle (twice) to deduce that $\neg\text{even}(m + n)$ holds. We leave some of the tedious details as an exercise. In particular, we use the fact proven in Chapter 2 that $\text{even}(p)$ iff $\neg\text{odd}(p)$ (see Section 2.16).

We treat *logical equivalence* as a derived connective: that is, we view $P \equiv Q$ as an abbreviation for $(P \Rightarrow Q) \wedge (Q \Rightarrow P)$. In view of the proof templates for $\wedge$, we see that to prove a logical equivalence $P \equiv Q$, we just have to prove both implications $P \Rightarrow Q$ and $Q \Rightarrow P$. For the sake of completeness, we state the following proof template.

Proof Template 1.13. *(Equivalence–Intro)*

Given a list of premises Γ, to obtain a deduction of an equivalence $P \equiv Q$, proceed as follows:

1. *Construct a deduction of the implication* $P \Rightarrow Q$ *from the list of premises* Γ.

2. *Construct a deduction of the implication* $Q \Rightarrow P$ *from the list of premises* Γ.

The proof templates described in this section and the previous one allow proving propositions which are known as the propositions of *classical propositional logic*. We also say that this set of proof templates is a *natural deduction proof system* for propositional logic; see Prawitz [41] and Gallier [14].

1.7 De Morgan Laws and Other Useful Rules of Logic

In Section 1.5, we proved certain implications that are special cases of the so-called *de Morgan laws*.

Proposition 1.1. *The following equivalences (de Morgan laws) are provable:*

$$\begin{aligned}\neg\neg P &\equiv P\\ \neg(P \wedge Q) &\equiv \neg P \vee \neg Q\\ \neg(P \vee Q) &\equiv \neg P \wedge \neg Q.\end{aligned}$$

The following equivalence expressing $\Rightarrow$ *in terms of* $\vee$ *and* $\neg$ *is also provable:*

$$P \Rightarrow Q \equiv \neg P \vee Q.$$

The following proposition (the law of the excluded middle) is provable:

$$P \vee \neg P.$$

The proofs that we have not shown are left as as exercises (sometimes tedious).

Proposition 1.1 shows a property that is very specific to classical logic, namely, that the logical connectives $\Rightarrow, \wedge, \vee, \neg$ are not independent. For example, we have $P \wedge Q \equiv \neg(\neg P \vee \neg Q)$, which shows that $\wedge$ can be expressed in terms of $\vee$ and $\neg$. Similarly, $P \Rightarrow Q \equiv \neg P \vee Q$ shows that $\Rightarrow$ can be expressed in terms of $\vee$ and $\neg$.

The next proposition collects a list of equivalences involving conjunction and disjunction that are used all the time. Constructing proofs using the proof templates is not hard but tedious.

Proposition 1.2. *The following propositions are provable:*

$$\begin{aligned}P \vee P &\equiv P\\ P \wedge P &\equiv P\\ P \vee Q &\equiv Q \vee P\\ P \wedge Q &\equiv Q \wedge P.\end{aligned}$$

The last two assert the commutativity of $\vee$ *and* $\wedge$*. We have distributivity of* $\wedge$ *over* $\vee$ *and of* $\vee$ *over* $\wedge$*:*

$$P \wedge (Q \vee R) \equiv (P \wedge Q) \vee (P \wedge R)$$
$$P \vee (Q \wedge R) \equiv (P \vee Q) \wedge (P \vee R).$$

We have associativity of $\wedge$ *and* $\vee$*:*

$$P \wedge (Q \wedge R) \equiv (P \wedge Q) \wedge R$$
$$P \vee (Q \vee R) \equiv (P \vee Q) \vee R.$$

1.8 Formal Versus Informal Proofs; Some Examples

In this section we give some explicit examples of proofs illustrating the proof templates that we just discussed. But first it should be said that *it is practically impossible to write formal proofs* (i.e., proofs written using the proof templates of the system presented earlier) of "real" statements that are not "toy propositions." This is because it would be extremely tedious and time-consuming to write such proofs and these proofs would be huge and thus very hard to read.

As we said before it is possible in principle to write formalized proofs, however, most of us will never do so. So what *do* we do?

Well, we construct "informal" proofs in which we still make use of the proof templates that we have presented but we take shortcuts and sometimes we even omit proof steps (some proof templates such as 1.9 (And–Elim) and 1.10 (Or–Intro)) and we use a natural language (here, presumably, English) rather than formal symbols (we say "and" for $\wedge$, "or" for $\vee$, etc.). As an example of a shortcut, when using the Proof Template 1.11 (Or–Elim), in most cases, the disjunction $P \vee Q$ has an "obvious proof" because P and Q "exhaust all the cases," in the sense that Q subsumes $\neg P$ (or P subsumes $\neg Q$) and classically, $P \vee \neg P$ is an axiom. Also, we implicitly keep track of the open premises of a proof in our head rather than explicitly delete premises when required. This may be the biggest source of mistakes and we should make sure that when we have finished a proof, there are no "dangling premises," that is, premises that were never used in constructing the proof. If we are "lucky," some of these premises are in fact unnecessary and we should discard them. Otherwise, this indicates that there is something wrong with our proof and we should make sure that every premise is indeed used somewhere in the proof or else look for a counterexample.

We urge our readers to read Chapter 3 of Gowers [26] which contains very illuminating remarks about the notion of proof in mathematics.

The next question is then, "How does one write good informal proofs?"

It is very hard to answer such a question because the notion of a "good" proof is quite subjective and partly a social concept. Nevertheless, people have been writing informal proofs for centuries so there are at least many examples of what to do (and what not to do). As with everything else, practicing a sport, playing a music instrument, knowing "good" wines, and so on, *the more you practice, the better you become*. Knowing the theory of swimming is fine but you have to get wet and do some actual swimming. Similarly, knowing the proof rules is important but you have to put them to use.

Write proofs as much as you can. Find good proof writers (like good swimmers, good tennis players, etc.), try to figure out why they write clear and easily readable proofs, and try to emulate what they do. Don't follow bad examples (it will take you a little while to "smell" a bad proof style).

Another important point is that nonformalized proofs make heavy use of *modus ponens*. This is because, when we search for a proof, we rarely (if ever) go back to first principles. This would result in extremely long proofs that would be basically incomprehensible. Instead, we search in our "database" of facts for a proposition of the form $P \Rightarrow Q$ (an auxiliary lemma) that is already known to be proven, and if we are smart enough (lucky enough), we find that we can prove P and thus we deduce Q, the proposition that we really want to prove. Generally, we have to go through several steps involving auxiliary lemmas. This is why it is important to build up a database of proven facts as large as possible about a mathematical field: numbers, trees, graphs, surfaces, and so on. This way we increase the chance that we will be able to prove some fact about some field of mathematics. practicing (constructing proofs).

And now we return to some explicit examples of informal proofs.

Recall that the *set of integers* is the set

$$\mathbb{Z} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$$

and that the *set of natural numbers* is the set

$$\mathbb{N} = \{0, 1, 2, \ldots\}.$$

(Some authors exclude 0 from $\mathbb{N}$. We don't like this discrimination against zero.) The following facts are essentially obvious from the definition of even and odd.

(a) The sum of even integers is even.

(b) The sum of an even integer and of an odd integer is odd.

(c) The sum of two odd integers is even.

(d) The product of odd integers is odd.

(e) The product of an even integer with any integer is even.

We will contruct deductions using sets of premises consisting of the above propositions.

Now we prove the following fact using the proof–by–cases method.

Proposition 1.3. *Let a, b, c be odd integers. For any integers p and q, if p and q are not both even, then*

$$ap^2 + bpq + cq^2$$

is odd.

Proof. We consider the three cases:

1. p and q are odd. In this case as a, b, and c are odd, by (d) all the products ap^2, bpq, and cq^2 are odd. By (c), $ap^2 + bpq$ is even and by (b), $ap^2 + bpq + cq^2$ is odd.

2. p is even and q is odd. In this case, by (e), both ap^2 and bpq are even and by (d), cq^2 is odd. But then, by (a), $ap^2 + bpq$ is even and by (b), $ap^2 + bpq + cq^2$ is odd.

3. p is odd and q is even. This case is analogous to the previous case, except that p and q are interchanged. The reader should have no trouble filling in the details.

All three cases exhaust all possibilities for p and q not to be both even, thus the proof is complete by Proof Template 1.11 applied twice, because there are three cases instead of two. □

The set of *rational numbers* $\mathbb{Q}$ consists of all fractions p/q, where $p, q \in \mathbb{Z}$, with $q \neq 0$. The set of real numbers is denoted by $\mathbb{R}$. A real number, $a \in \mathbb{R}$, is said to be *irrational* if it cannot be expressed as a number in $\mathbb{Q}$ (a fraction).

We now use Proposition 1.3 and the proof by contradiction method to prove the following.

Proposition 1.4. *Let a, b, c be odd integers. Then the equation*

$$aX^2 + bX + c = 0$$

has no rational solution X. Equivalently, every zero of the above equation is irrational.

Proof. We proceed by contradiction (by this we mean that we use the proof–by–contradiction principle). So assume that there is a rational solution $X = p/q$. We may assume that p and q have no common divisor, which implies that p and q are not both even. As $q \neq 0$, if $aX^2 + bX + c = 0$, then by multiplying by q^2, we get

$$ap^2 + bpq + cq^2 = 0.$$

However, as p and q are not both even and a, b, c are odd, we know from Proposition 1.3 that $ap^2 + bpq + cq^2$ is odd. This contradicts the fact that $p^2 + bpq + cq^2 = 0$ and thus finishes the proof. □

As as example of the proof–by–contrapositive method, we prove that if an integer n^2 is even, then n must be even.

Observe that if an integer is not even then it is odd (and vice versa). This fact may seem quite obvious but to prove it actually requires using *induction* (which we haven't officially met yet). A rigorous proof is given in Section 1.12.

Now the contrapositive of our statement is: if n is odd, then n^2 is odd. But, to say that n is odd is to say that $n = 2k + 1$ and then, $n^2 = (2k + 1)^2 = 4k^2 + 4k + 1 = 2(2k^2 + 2k) + 1$, which shows that n^2 is odd.

As another illustration of the proof methods that we have just presented, let us prove that $\sqrt{2}$ is irrational, which means that $\sqrt{2}$ is *not* rational. The reader may also want to look at the proof given by Gowers in Chapter 3 of his book [26]. Obviously, our proof is similar but we emphasize step (2) a little more.

Because we are trying to prove that $\sqrt{2}$ is not rational, we use Proof Template 1.4. Thus let us assume that $\sqrt{2}$ is rational and derive a contradiction. Here are the steps of the proof.

1. If $\sqrt{2}$ is rational, then there exist some integers $p, q \in \mathbb{Z}$, with $q \neq 0$, so that $\sqrt{2} = p/q$.

2. Any fraction p/q is equal to some fraction r/s, where r and s are not both even.

3. By (2), we may assume that
$$\sqrt{2} = \frac{p}{q},$$
where $p, q \in \mathbb{Z}$ are *not both even* and with $q \neq 0$.

4. By (3), because $q \neq 0$, by multiplying both sides by q, we get
$$q\sqrt{2} = p.$$

5. By (4), by squaring both sides, we get
$$2q^2 = p^2.$$

6. Inasmuch as $p^2 = 2q^2$, the number p^2 must be even. By a fact previously established, *p itself is even*; that is, $p = 2s$, for some $s \in \mathbb{Z}$.

7. By (6), if we substitute $2s$ for p in the equation in (5) we get $2q^2 = 4s^2$. By dividing both sides by 2, we get
$$q^2 = 2s^2.$$

8. By (7), we see that q^2 is even, from which we deduce (as above) that *q itself is even*.

9. Now, assuming that $\sqrt{2} = p/q$ where p and q are *not both even* (and $q \neq 0$), we concluded that *both p and q are even* (as shown in (6) and(8)), reaching a contradiction. Therefore, by negation introduction, we proved that $\sqrt{2}$ is *not* rational.

A closer examination of the steps of the above proof reveals that the only step that may require further justification is step (2): that any fraction p/q is equal to some fraction r/s where r and s are not both even.

This fact does require a proof and the proof uses the division algorithm, which itself requires induction. Besides this point, all the other steps only require simple arithmetic properties of the integers and are constructive.

Remark: Actually, every fraction p/q is equal to some fraction r/s where r and s have no common divisor except 1. This follows from the fact that every pair of integers has a *greatest* *common divisor* (a *gcd*; s and r and s are obtained by dividing p and q by their gcd. Using this fact and Euclid's lemma, we can obtain a shorter proof of the irrationality of $\sqrt{2}$. First we may assume that p and q have no common divisor besides 1 (we say that p and q are *relatively prime*). From (5), we have

$$2q^2 = p^2,$$

so q divides p^2. However, q and p are relatively prime and as q divides $p^2 = p \times p$, by Euclid's lemma, q divides p. But because 1 is the only common divisor of p and q, we must have $q = 1$. Now, we get $p^2 = 2$, which is impossible inasmuch as 2 is not a perfect square.

The above argument can be easily adapted to prove that if the positive integer n is not a perfect square, then $\sqrt{n}$ is not rational.

We conclude this section by showing that the proof–by–contradiction principle allows for proofs of propositions that may lack a constructive nature. In particular, *it is possible to prove disjunctions $P \vee Q$ which states some alternative that cannot be settled*.

For example, consider the question: are there two irrational real numbers a and b such that a^b is rational? Here is a way to prove that this is indeed the case. Consider the number $\sqrt{2}^{\sqrt{2}}$. If this number is rational, then $a = \sqrt{2}$ and $b = \sqrt{2}$ is an answer to our question (because we already know that $\sqrt{2}$ is irrational). Now observe that

$$(\sqrt{2}^{\sqrt{2}})^{\sqrt{2}} = \sqrt{2}^{\sqrt{2}\times\sqrt{2}} = \sqrt{2}^2 = 2 \quad \text{is rational.}$$

Thus, if $\sqrt{2}^{\sqrt{2}}$ is not rational, then $a = \sqrt{2}^{\sqrt{2}}$ and $b = \sqrt{2}$ is an answer to our question. Because $P \vee \neg P$ is provable *using the proof–by–contradiction principle* ($\sqrt{2}^{\sqrt{2}}$ is rational or it is not rational), we proved that

($\sqrt{2}$ is irrational and $\sqrt{2}^{\sqrt{2}}$ is rational) or

($\sqrt{2}^{\sqrt{2}}$ and $\sqrt{2}$ are irrational and $(\sqrt{2}^{\sqrt{2}})^{\sqrt{2}}$ is rational).

However, the above proof does not tell us whether $\sqrt{2}^{\sqrt{2}}$ is rational!

We see one of the shortcomings of classical reasoning: certain statements (in particular, disjunctive or existential) are provable but their proof does not provide an explicit answer. For this reason, classical logic is considered to be nonconstructive.

Remark: Actually, it turns out that another irrational number b can be found so that $\sqrt{2}^b$ is rational and the proof that b is not rational is fairly simple. It also turns out that the exact nature of $\sqrt{2}^{\sqrt{2}}$ (rational or irrational) is known. The answers to these puzzles can be found in Section 1.10.

1.9 Truth Tables and Truth Value Semantics

So far we have deliberately focused on the construction of proofs using proof templates, we but have ignored the notion of truth. We can't postpone any longer a discussion of the truth value semantics for classical propositional logic.

We all learned early on that the logical connectives $\Rightarrow$, $\wedge$, $\vee$, $\neg$ and $\equiv$ can be interpreted as Boolean functions, that is, functions whose arguments and whose values range over the set of *truth values*,

$$\mathbf{BOOL} = \{\mathbf{true}, \mathbf{false}\}.$$

These functions are given by the following *truth tables*.

P	Q	$P \Rightarrow Q$	$P \wedge Q$	$P \vee Q$	$\neg P$	$P \equiv Q$
true	**true**	**true**	**true**	**true**	**false**	**true**
true	**false**	**false**	**false**	**true**	**false**	**false**
false	**true**	**true**	**false**	**true**	**true**	**false**
false	**false**	**true**	**false**	**false**	**true**	**true**

Note that the implication $P \Rightarrow Q$ is false (has the value **false**) exactly when $P = \mathbf{true}$ and $Q = \mathbf{false}$.

Now any proposition P built up over the set of atomic propositions **PS** (our propositional symbols) contains a finite set of propositional letters, say

$$\{P_1, \ldots, P_m\}.$$

If we assign some truth value (from **BOOL**) to each symbol P_i then we can "compute" the *truth value* of P under this assignment by using recursively using the truth tables above. For example, the proposition $\mathbf{P}_1 \Rightarrow (\mathbf{P}_1 \Rightarrow \mathbf{P}_2)$, under the truth assignment v given by

$$\mathbf{P}_1 = \mathbf{true},\ \mathbf{P}_2 = \mathbf{false},$$

evaluates to **false**. Indeed, the truth value, $v(\mathbf{P}_1 \Rightarrow (\mathbf{P}_1 \Rightarrow \mathbf{P}_2))$, is computed recursively as

$$v(\mathbf{P}_1 \Rightarrow (\mathbf{P}_1 \Rightarrow \mathbf{P}_2)) = v(\mathbf{P}_1) \Rightarrow v(\mathbf{P}_1 \Rightarrow \mathbf{P}_2).$$

Now, $v(\mathbf{P}_1) = \mathbf{true}$ and $v(\mathbf{P}_1 \Rightarrow \mathbf{P}_2)$ is computed recursively as

$$v(\mathbf{P}_1 \Rightarrow \mathbf{P}_2) = v(\mathbf{P}_1) \Rightarrow v(\mathbf{P}_2).$$

Because $v(\mathbf{P}_1) = \mathbf{true}$ and $v(\mathbf{P}_2) = \mathbf{false}$, using our truth table, we get

$$v(\mathbf{P}_1 \Rightarrow \mathbf{P}_2) = \mathbf{true} \Rightarrow \mathbf{false} = \mathbf{false}.$$

Plugging this into the right-hand side of $v(\mathbf{P}_1 \Rightarrow (\mathbf{P}_1 \Rightarrow \mathbf{P}_2))$, we finally get

$$v(\mathbf{P}_1 \Rightarrow (\mathbf{P}_1 \Rightarrow \mathbf{P}_2)) = \mathbf{true} \Rightarrow \mathbf{false} = \mathbf{false}.$$

However, under the truth assignment v given by

$$\mathbf{P}_1 = \mathbf{true},\ \mathbf{P}_2 = \mathbf{true},$$

we find that our proposition evaluates to **true**.

The values of a proposition can be determined by creating a *truth table*, in which a proposition is evaluated by computing recursively the truth values of its subexpressions. For example, the truth table corresponding to the proposition $\mathbf{P}_1 \Rightarrow (\mathbf{P}_1 \Rightarrow \mathbf{P}_2)$ is

$\mathbf{P}_1$	$\mathbf{P}_2$	$\mathbf{P}_1 \Rightarrow \mathbf{P}_2$	$\mathbf{P}_1 \Rightarrow (\mathbf{P}_1 \Rightarrow \mathbf{P}_2)$
true	**true**	**true**	**true**
true	**false**	**false**	**false**
false	**true**	**true**	**true**
false	**false**	**true**	**true**

If we now consider the proposition $P = (\mathbf{P}_1 \Rightarrow (\mathbf{P}_2 \Rightarrow \mathbf{P}_1))$, its truth table is

$\mathbf{P}_1$	$\mathbf{P}_2$	$\mathbf{P}_2 \Rightarrow \mathbf{P}_1$	$\mathbf{P}_1 \Rightarrow (\mathbf{P}_2 \Rightarrow \mathbf{P}_1)$
true	**true**	**true**	**true**
true	**false**	**true**	**true**
false	**true**	**false**	**true**
false	**false**	**true**	**true**

which shows that P evaluates to **true** for all possible truth assignments.

The truth table of a proposition containing m variables has 2^m rows. When m is large, 2^m is very large, and computing the truth table of a proposition P may not be practically feasible. Even the problem of finding whether there is a truth assignment that makes P true is hard. This is actually a very famous problem in computer science.

A proposition P is said to be *valid* or a *tautology* if in the truth table for P all the entries in the column corresponding to P have the value **true**. This means that P evaluates to **true** for all 2^m truth assignments.

What's the relationship between validity and provability? *Remarkably, validity and provability are equivalent.*

In order to prove the above claim, we need to do two things:

(1) Prove that if a proposition P is provable using the proof templates that we described earlier, then it is valid. This is known as *soundness* or *consistency* (of the proof system).

(2) Prove that if a proposition P is valid, then it has a proof using the proof templates. This is known as the *completeness* (of the proof system).

In general, it is relatively easy to prove (1) but proving (2) can be quite complicated.

In this book we content ourselves with soundness.

Proposition 1.5. *(Soundness of the proof templates) If a proposition P is provable using the proof templates desribed earlier, then it is valid (according to the truth value semantics).*

Sketch of Proof. It is enough to prove that if there is a deduction of a proposition P from a set of premises Γ, then for every truth assignment for which all the propositions in Γ evaluate to **true**, then P evaluates to **true**. However, this is clear for the axioms and every proof template preserves that property.

Now, if P is provable, a proof of P has an empty set of premises and so P evaluates to **true** for all truth assignments, which means that P is valid. $\square$

Theorem 1.6. *(Completeness) If a proposition P is valid (according to the truth value semantics), then P is provable using the proof templates.*

Proofs of completeness for classical logic can be found in van Dalen [51] or Gallier [19] (but for a different proof system).

Soundness (Proposition 1.5) has a very useful consequence: in order to prove that a proposition P is *not provable*, it is enough to find a truth assignment for which P evaluates to **false**. We say that such a truth assignment is a *counterexample* for P (or that P can be *falsified*).

For example, no propositional symbol $\mathbf{P}_i$ is provable because it is falsified by the truth assignment $\mathbf{P}_i = \mathbf{false}$.

The soundness of our proof system also has the extremely important consequence that $\perp$ *cannot be proven* in this system, which means that *contradictory statements* cannot be derived.

This is by no means obvious at first sight, but reassuring.

Note that completeness amounts to the fact that every unprovable proposition has a counterexample. Also, in order to show that a proposition is provable, it suffices to compute its truth table and check that the proposition is valid. This may still be a lot of work, but it is a more "mechanical" process than attempting to find a proof. For example, here is a truth table showing that $(\mathbf{P}_1 \Rightarrow \mathbf{P}_2) \equiv (\neg \mathbf{P}_1 \vee \mathbf{P}_2)$ is valid.

$\mathbf{P}_1$	$\mathbf{P}_2$	$\mathbf{P}_1 \Rightarrow \mathbf{P}_2$	$\neg \mathbf{P}_1 \vee \mathbf{P}_2$	$(\mathbf{P}_1 \Rightarrow \mathbf{P}_2) \equiv (\neg \mathbf{P}_1 \vee \mathbf{P}_2)$
true	**true**	**true**	**true**	**true**
true	**false**	**false**	**false**	**true**
false	**true**	**true**	**true**	**true**
false	**false**	**true**	**true**	**true**

1.10 Proof Templates for the Quantifiers

As we mentioned in Section 1.1, atomic propositions may contain variables. The intention is that such variables correspond to arbitrary objects. An example is

$$\text{human}(x) \Rightarrow \text{needs-to-drink}(x).$$

In mathematics, we usually prove universal statements, that is statements that hold for all possible "objects," or existential statements, that is, statements asserting the existence of some object satisfying a given property. As we saw earlier, we assert that every human needs to drink by writing the proposition

$$\forall x(\text{human}(x) \Rightarrow \text{needs-to-drink}(x)).$$

The symbol $\forall$ is called a *universal quantifier*. Observe that once the quantifier $\forall$ (pronounced "for all" or "for every") is applied to the variable x, the variable x becomes a placeholder and replacing x by y or any other variable *does not change anything*. We say that x is a *bound variable* (sometimes a "dummy variable").

If we want to assert that some human needs to drink we write

$$\exists x(\text{human}(x) \Rightarrow \text{needs-to-drink}(x));$$

The symbol $\exists$ is called an *existential quantifier*. Again, once the quantifier $\exists$ (pronounced "there exists") is applied to the variable x, the variable x becomes a placeholder. However, the intended meaning of the second proposition is very different and weaker than the first. It only asserts the existence of some object satisfying the statement

$$\text{human}(x) \Rightarrow \text{needs-to-drink}(x).$$

Statements may contain variables that are not bound by quantifiers. For example, in

$$\exists x \, \text{parent}(x, y)$$

the variable x is bound but the variable y is not. Here, the intended meaning of $\text{parent}(x, y)$ is that x is a parent of y, and the intended meaning of $\exists x \, \text{parent}(x, y)$ is that any given y has some parent x. Variables that are not bound are called *free*. The proposition

$$\forall y \exists x \, \text{parent}(x, y),$$

which contains only bound variables is meant to assert that every y has some parent x. Typically, in mathematics, we only prove statements without free variables. However, statements with free variables may occur during intermediate stages of a proof.

Now, in addition to propositions of the form $P \wedge Q, P \vee Q, P \Rightarrow Q, \neg P, P \equiv Q$, we add two new kinds of propositions (also called formulae):

1. *Universal formulae*, which are formulae of the form $\forall x P$, where P is any formula and x is any variable.

2. *Existential formulae*, which are formulae of the form $\exists x P$, where P is any formula and x is any variable.

The intuitive meaning of the statement $\forall x P$ is that P holds for all possible objects x and the intuitive meaning of the statement $\exists x P$ is that P holds for some object x. Thus we see that it would be useful to use symbols to denote various objects. For example, if we want to assert some facts about the "parent" predicate, we may want to introduce some *constant symbols* (for short, constants) such as "Jean," "Mia," and so on and write

$$\text{parent}(\text{Jean}, \text{Mia})$$

to assert that Jean is a parent of Mia. Often we also have to use *function symbols* (or *operators, constructors*), for instance, to write a statement about numbers: $+$, $*$, and so on. Using constant symbols, function symbols, and variables, we can form *terms*, such as

$$(x * x + 1) * (3 * y + 2).$$

In addition to function symbols, we also use *predicate symbols*, which are names for atomic properties. We have already seen several examples of predicate symbols: "odd," "even," "prime," "human," "parent." So in general, when we try to prove properties of certain classes of objects (people, numbers, strings, graphs, and so on), we assume that we have a certain *alphabet* consisting of constant symbols, function symbols, and predicate symbols. Using these symbols and an infinite supply of variables we can form *terms* and *predicate terms*. We say that we have a *(logical) language*. Using this language, we can write compound statements. A detailed presentation of this approach is given in Chapter 2. Here we follow a more informal and more intuitive approach. We use the notion of term as a synonym for some specific object. Terms are often denoted by the Greek letter τ, sometimes subscripted. A variable qualifies as a term.

When working with propositions possibly containing quantifiers, it is customary to use the term *formula* instead of proposition. The term proposition is typically reserved to formulae wihout quantifiers.

Unlike the Proof Templates for $\Rightarrow, \vee, \wedge$ and $\perp$, which are rather straightforward, the Proof Templates for quantifiers are more subtle due to the presence of variables (occurring in terms and predicates) and the fact that it is sometimes necessary to make *substitutions*.

Given a formula P containing some free variable x and given a term τ, the result of replacing all occurrences of x by τ in P is called a *substitution* and is denoted $P[\tau/x]$ (and pronounced "the result of substituting τ for x in P"). Substitutions can be defined rigorously by recursion. Let us simply give an example. Consider the predicate $P(x) = \text{odd}(2x + 1)$. If we substitute the term $\tau = (y + 1)^2$ for x in $P(x)$, we obtain

$$P[\tau/x] = \text{odd}(2(y + 1)^2 + 1).$$

We have to be careful to forbid inferences that would yield "wrong" results and for this we have to be very precise about the way we use free variables. More specifically, we have to exercise care when we make *substitutions* of terms for variables in propositions. If $P(t_1, t_2, \ldots, t_n)$ is a statement containing the free variables $t_1, \ldots, t_n$ and if $\tau_1, \ldots, \tau_n$ are terms, we can form the new statement

$$P[\tau_1/t_1, \ldots, \tau_n/t_n]$$

obtained by substituting the term τ_i for all free occurrences of the variable t_i, for $i = 1, \ldots, n$. By the way, we denote terms by the Greek letter τ because we use the letter t for a variable and using t for both variables and terms would be confusing; sorry.

However, if $P(t_1, t_2, \ldots, t_n)$ contains quantifiers, some bad things can happen; namely, some of the variables occurring in some term τ_i may become quantified when τ_i is substituted for t_i. For example, consider

$$\forall x \exists y\, P(x, y, z)$$

which contains the free variable z and substitute the term $x + y$ for z: we get

$$\forall x \exists y\, P(x, y, x + y).$$

We see that the variables x and y occurring in the term $x + y$ become bound variables after substitution. We say that there is a "capture" of variables.

This is not what we intended to happen. To fix this problem, we recall that bound variables are really place holders so they can be renamed without changing anything. Therefore, we can rename the bound variables x and y in $\forall x \exists y\, P(x, y, z)$ to u and v, getting the statement $\forall u \exists v\, P(u, v, z)$ and now, the result of the substitution is

$$\forall u \exists v\, P(u, v, x + y),$$

where x and y are free. Again, all this needs to be explained very carefuly but in this chapter we will content ourselves with an informal treatment.

We begin with the proof templates for the universal quantifier.

Proof Template 1.14. *(Forall–Intro)*

Let Γ be a list of premises and let y be a variable that does not occur free in any premise in Γ or in $\forall x P$. If we have a deduction of the formula $P[y/x]$ from Γ, then we obtain a deduction of $\forall x P$ from Γ.

Proof Template 1.15. *(Forall–Elim)*

Let Γ be a list of premises and let τ be a term representing some specific object. If we have a deduction of $\forall x P$ from Γ, then we obtain a deduction of $P[\tau/x]$ from Γ.

The Proof Template 1.14 may look a little strange but the idea behind it is actually very simple: Because y is totally unconstrained, if $P[y/x]$ (the result of replacing all occurrences of x by y in P) is provable (from Γ), then intuitively $P[y/x]$ holds for any arbitrary object, and so, the statement $\forall x P$ should also be provable (from Γ).

Note that we can't deduce $\forall x P$ from $P[y/x]$ because the deduction has the single premise $P[y/x]$ and y occurs in $P[y/x]$ (unless x does not occur in P).

The meaning of the Proof Template 1.15 is that if $\forall x P$ is provable (from Γ), then P holds for all objects and so, in particular for the object denoted by the term τ; that is, $P[\tau/x]$ should be provable (from Γ).

Here are the proof templates for the existential quantifier.

Proof Template 1.16. *(Exist–Intro)*

Let Γ be a list of premises and let τ be a term representing some specific object. If we have a deduction of $P[\tau/x]$ from Γ, then we obtain a deduction of $\exists x P(x)$ from Γ.

Proof Template 1.17. *(Exist–Elim)*

Let Γ and Δ be a two lists of premises. Let C and $\exists x P$ be formulae, and let y be a variable that does not occur free in any premise in Γ, in $\exists x P$, or in C. To obtain a deduction of C from Γ, Δ, proceed as follows:

1. *Make a deduction of $\exists x P$ from Γ.*

2. *Add one or more occurrences of $P[y/x]$ as premises to Δ, and find a deduction of C from $P[y/x]$ and Δ.*

3. *Delete the premise $P[y/x]$.*

If $P[\tau/x]$ is provable (from Γ), this means that the object denoted by τ satisfies P, so $\exists x P$ should be provable (this latter formula asserts the existence of some object satisfying P, and τ is such an object).

Proof Template 1.17 is reminiscent of the proof–by–cases principle (Proof template 1.11) and is a little more tricky. It goes as follows. Suppose that we proved $\exists x P$ (from Γ). Moreover, suppose that for every possible case $P[y/x]$ we were able to prove C (from Δ). Then, as we have "exhausted" all possible cases and as we know from the provability of $\exists x P$ that some case must hold, we can conclude that C is provable (from Γ, Δ) without using $P[y/x]$ as a premise.

Like the the proof–by–cases principle, Proof Template 1.17 is not very constructive. It allows making a conclusion (C) by considering alternatives without knowing which one actually occurs.

Constructing proofs using the proof templates for the quantifiers can be quite tricky due to the restrictions on variables. In practice, we always use "fresh" (brand new) variables to avoid problems. Also, when we use Proof Template 1.14, we begin by saying "let y be arbitrary," then we prove $P[y/x]$ (mentally substituting y for x), and we conclude with: "since y is arbitrary, this proves $\forall x P$." We proceed in a similar way when using Proof Template 1.17, but this time we say "let y be arbitrary" in step (2). When we use Proof Template 1.15, we usually say: "Since $\forall x P$ holds, it holds for all x, so in particular it holds for τ, and thus $P[\tau/x]$ holds." Similarly, when using Proof Template 1.16, we say "since $P[\tau/x]$ holds for a specific object τ, we can deduce that $\exists x P$ holds."

Here is an example of a "wrong proof" in which the $\forall$-introduction rule is applied illegally, and thus, yields a statement that is actually false (not provable). In the incorrect "proof" below, P is an atomic predicate symbol taking two arguments (e.g., "parent") and 0 is a constant denoting zero:

$$\cfrac{\cfrac{\cfrac{\cfrac{P(u,0)^x}{\forall t P(t,0)}\ \text{illegal step!}}{P(u,0) \Rightarrow \forall t P(t,0)}\ \text{Implication-Intro}\ x}{\forall s(P(s,0) \Rightarrow \forall t P(t,0))}\ \text{Forall-Intro}}{P(0,0) \Rightarrow \forall t P(t,0)}\ \text{Forall-Elim}$$

The problem is that the variable u occurs free in the premise $P[u/t,0] = P(u,0)$ and therefore, *the application of the $\forall$-introduction rule in the first step is illegal*. However, note that this premise is discharged in the second step and so, the application of the $\forall$-introduction rule in the third step is legal. The (false) conclusion of this faulty proof is that $P(0,0) \Rightarrow \forall t P(t,0)$ is provable. Indeed, there are plenty of properties such that the fact that the single instance $P(0,0)$ holds does not imply that $P(t,0)$ holds for all t.

Let us now give two examples of a proof using the proof templates for $\forall$ and $\exists$.

Example 1.21. For any natural number n, let $\text{odd}(n)$ be the predicate that asserts that n is odd, namely

$$\text{odd}(n) \equiv \exists m((m \in \mathbb{N}) \wedge (n = 2m + 1)).$$

First let us prove that

$$\forall a((a \in \mathbb{N}) \Rightarrow \text{odd}(2a + 1)).$$

By Proof Template 1.14, let x be a fresh variable; we need to prove

$$(x \in \mathbb{N}) \Rightarrow \text{odd}(2x + 1).$$

By Proof Template 1.2, assume $x \in \mathbb{N}$. If we consider the formula

$$(m \in \mathbb{N}) \wedge (2x + 1 = 2m + 1),$$

by substituting x for m, we get

$$(x \in \mathbb{N}) \wedge (2x + 1 = 2x + 1),$$

which is provable since $x \in \mathbb{N}$. By Proof Template 1.16, we obtain

$$\exists m(m \in \mathbb{N}) \wedge (2x + 1 = 2m + 1);$$

that is, $\text{odd}(2x+1)$ is provable. Using Proof Template 1.2, we delete the premise $x \in \mathbb{N}$ and we have proven

$$(x \in \mathbb{N}) \Rightarrow \text{odd}(2x + 1).$$

This proof has no longer any premises, so we can safely conclude that

$$\forall a((a \in \mathbb{N}) \Rightarrow \text{odd}(2a + 1)).$$

Next consider the term $\tau = 7$. By Proof Template 1.15, we obtain

$$(7 \in \mathbb{N}) \Rightarrow \mathrm{odd}(15).$$

Since $7 \in \mathbb{N}$, by modus ponens we deduce that 15 is odd.

Let us now consider the term $\tau = (b+1)^2$ with $b \in \mathbb{N}$. By Proof Template 1.15, we obtain

$$((b+1)^2 \in \mathbb{N}) \Rightarrow \mathrm{odd}(2(b+1)^2 + 1)).$$

But $b \in \mathbb{N}$ implies that $(b+1)^2 \in \mathbb{N}$ so by modus ponens and Proof Template 1.2, we deduce that

$$(b \in \mathbb{N}) \Rightarrow \mathrm{odd}(2(b+1)^2 + 1)).$$

Example 1.22. Let us prove the formula $\forall x(P \wedge Q) \Rightarrow \forall xP \wedge \forall xQ$.

First using Proof Template 1.2, we assume $\forall x(P \wedge Q)$ (two copies). The next step uses a trick. Since variables are terms, if u is a fresh variable, then by Proof Templare 1.15 we deduce $(P \wedge Q)[u/x]$. Now we use a property of substitutions which says that

$$(P \wedge Q)[u/x] = P[u/x] \wedge Q[u/x].$$

We can now use Proof Template 1.9 (twice) to deduce $P[u/x]$ and $Q[u/x]$. But, remember that the premise is $\forall x(P \wedge Q)$ (two copies), and since u is a fresh variable, it does not occur in this premise, so we can safely apply Proof Template 1.14 and conclude $\forall xP$, and similarly $\forall xQ$. By Proof Template 1.8, we deduce $\forall xP \wedge \forall xQ$ from $\forall x(P \wedge Q)$. Finally, by Proof Template 1.2, we delete the premise $\forall x(P \wedge Q)$ and obtain our proof. The above proof has the following tree representation.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\forall x(P \wedge Q)^{x\checkmark}}{P[u/x] \wedge Q[u/x]}}{P[u/x]}}{\forall xP} \qquad \dfrac{\dfrac{\dfrac{\forall x(P \wedge Q)^{x\checkmark}}{P[u/x] \wedge Q[u/x]}}{Q[u/x]}}{\forall xQ}}{\forall xP \wedge \forall xQ}}{\forall x(P \wedge Q) \Rightarrow \forall xP \wedge \forall xQ}\; x$$

The reader should show that $\forall xP \wedge \forall xQ \Rightarrow \forall x(P \wedge Q)$ is also provable.

However, in general, one can't just replace $\forall$ by $\exists$ (or $\wedge$ by $\vee$) and still obtain provable statements. For example, $\exists xP \wedge \exists xQ \Rightarrow \exists x(P \wedge Q)$ is not provable at all.

Here are some useful equivalences involving quantifiers. The first two are analogous to the de Morgan laws for $\wedge$ and $\vee$.

Proposition 1.7. *The following formulae are provable:*

$$\begin{aligned}\neg\forall xP &\equiv \exists x\neg P\\ \neg\exists xP &\equiv \forall x\neg P\\ \forall x(P\wedge Q) &\equiv \forall xP\wedge\forall xQ\\ \exists x(P\vee Q) &\equiv \exists xP\vee\exists xQ\\ \exists x(P\wedge Q) &\Rightarrow \exists xP\wedge\exists xQ\\ \forall xP\vee\forall xQ &\Rightarrow \forall x(P\vee Q).\end{aligned}$$

The proof system that uses all the Proof Templates that we have defined proves formulae of *classical first-order logic*.

One should also be careful that the order the quantifiers is important. For example, a formula of the form

$$\forall x\exists yP$$

is generally not equivalent to the formula

$$\exists y\forall xP.$$

The second formula asserts the existence of some object y such that P holds for all x. But in the first formula, for every x, there is some y such that P holds, but each y depends on x and *there may not be a single y that works for all x*.

Another amusing mistake involves negating a universal quantifier. The formula $\forall x\neg P$ is not equivalent to $\neg\forall xP$. Once traveling from Philadelphia to New York I heard a train conductor say: "all doors will not open." Actually, he meant "not all doors will open," which would give us a chance to get out!

Remark: We can illustrate, again, the fact that classical logic allows for nonconstructive proofs by re-examining the example at the end of Section 1.5. There we proved that if $\sqrt{2}^{\sqrt{2}}$ is rational, then $a=\sqrt{2}$ and $b=\sqrt{2}$ are both irrational numbers such that a^b is rational and if $\sqrt{2}^{\sqrt{2}}$ is irrational then $a=\sqrt{2}^{\sqrt{2}}$ and $b=\sqrt{2}$ are both irrational numbers such that a^b is rational. By Proof Template 1.16, we deduce that if $\sqrt{2}^{\sqrt{2}}$ is rational, then there exist some irrational numbers a,b so that a^b is rational, and if $\sqrt{2}^{\sqrt{2}}$ is irrational, then there exist some irrational numbers a,b so that a^b is rational. In classical logic, as $P\vee\neg P$ is provable, by the proof–by–cases principle we just proved that there exist some irrational numbers a and b so that a^b is rational.

However, this argument does not give us explicitly numbers a and b with the required properties. It only tells us that such numbers must exist.

Now, it turns out that $\sqrt{2}^{\sqrt{2}}$ is indeed irrational (this follows from the Gel'fond–Schneider theorem, a hard theorem in number theory). Furthermore, there are also simpler explicit solutions such as $a=\sqrt{2}$ and $b=\log_2 9$, as the reader should check.

1.11 Sets and Set Operations

In this section we review the definition of a set and basic set operations. This section takes the "naive" point of view that a set is an unordered collection of objects, without duplicates, the collection being regarded as a single object.

Given a set A we write that some object a is an element of (belongs to) the set A as

$$a \in A$$

and that a is not an element of A (does not belong to A) as

$$a \notin A.$$

The symbol $\in$ is the *set membership* symbol.

A set can either be defined explicitely by listing its elements within curly braces (the symbols $\{$ and $\}$) or as a collection of objects satisfying a certain property. For example, the set C consisting of the colors red, blue, green is given by

$$C = \{\text{red}, \text{blue}, \text{green}\}.$$

Because the order of elements in a set is irrelevant, the set C is also given by

$$C = \{\text{green}, \text{red}, \text{blue}\}.$$

In fact, a moment of reflexion reveals that there are six ways of writing the set C.

If we denote by $\mathbb{N}$ the set of natural numbers

$$\mathbb{N} = \{0, 1, 2, 3, \ldots\},$$

then the set E of even integers can be defined in terms of the property even of being even by

$$E = \{n \in \mathbb{N} \mid \text{even}(n)\}.$$

More generally, given some property P and some set X, we denote the set of all elements of X that satisfy the property P by

$$\{x \in X \mid P(x)\} \quad \text{or} \quad \{x \mid x \in X \wedge P(x)\}.$$

When are two sets A and B equal? The answer is given by the first proof template of set theory, called the Extensionality Axiom.

Proof Template 1.18. *(Extensionality Axiom)*

Two sets A and B are equal iff they have exactly the same elements; that is, every element of A is an element of B and conversely. This can be written more formally as

$$\forall x(x \in A \Rightarrow x \in B) \wedge \forall x(x \in B \Rightarrow x \in A).$$

There is a special set having no elements at all, the *empty set*, denoted $\emptyset$. The empty set is characterized by the property

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

Next we define the notion of inclusion between sets

Definition 1.5. Given any two sets, A and B, we say that A *is a subset of* B (or that A *is included in* B), denoted $A \subseteq B$, iff every element of A is also an element of B, that is,

$$\forall x(x \in A \Rightarrow x \in B).$$

We say that A *is a proper subset of* B iff $A \subseteq B$ and $A \neq B$. This implies that that there is some $b \in B$ with $b \notin A$. We usually write $A \subset B$.

For example, if $A = \{\text{green}, \text{blue}\}$ and $C = \{\text{green}, \text{red}, \text{blue}\}$, then

$$A \subseteq C.$$

Note that the empty set is a subset of every set.

Observe the important fact that equality of two sets can be expressed by

$$A = B \quad \text{iff} \quad A \subseteq B \quad \text{and} \quad B \subseteq A.$$

Proving that two sets are equal may be quite complicated if the definitions of these sets are complex, and the above method is the safe one.

If a set A has a finite number of elements, then this number (a natural number) is called the *cardinality* of the set and is denoted by $|A|$ (sometimes by $\text{card}(A)$). Otherwise, the set is said to be *infinite*. The cardinality of the empty set is 0.

Sets can be combined in various ways, just as numbers can be added, multiplied, *etc.* However, operations on sets tend to minic logical operations such as disjunction, conjunction, and negation, rather than the arithmetical operations on numbers. The most basic operations are union, intersection, and relative complement.

Definition 1.6. For any two sets A and B, the *union of* A *and* B is the set $A \cup B$ defined such that

$$x \in A \cup B \quad \text{iff} \quad (x \in A) \vee (x \in B).$$

This reads, x is a member of $A \cup B$ if either x belongs to A or x belongs to B (or both). We also write

$$A \cup B = \{x \mid x \in A \quad \text{or} \quad x \in B\}.$$

The *intersection of* A *and* B is the set $A \cap B$ defined such that

$$x \in A \cap B \quad \text{iff} \quad (x \in A) \wedge (x \in B).$$

This reads, x is a member of $A \cap B$ if x belongs to A and x belongs to B. We also write

$$A \cap B = \{x \mid x \in A \quad \text{and} \quad x \in B\}.$$

The *relative complement (or set difference) of A and B* is the set $A - B$ defined such that

$$x \in A - B \quad \text{iff} \quad (x \in A) \wedge \neg(x \in B).$$

This reads, x is a member of $A - B$ if x belongs to A and x does not belong to B. We also write

$$A - B = \{x \mid x \in A \quad \text{and} \quad x \notin B\}.$$

For example, if $A = \{0, 2, 4, 6\}$ and $B = \{0, 1, 3, 5\}$, then

$$\begin{aligned} A \cup B &= \{0, 1, 2, 3, 4, 5, 6\} \\ A \cap B &= \{0\} \\ A - B &= \{2, 4, 6\}. \end{aligned}$$

Two sets A, B are said to be *disjoint* if $A \cap B = \emptyset$. It is easy to see that if A and B are two finite sets and if A and B are disjoint, then

$$|A \cup B| = |A| + |B|.$$

In general, by writing

$$A \cup B = (A \cap B) \cup (A - B) \cup (B - A),$$

if A and B are finite, it can be shown that

$$|A \cup B| = |A| + |B| - |A \cap B|.$$

The situation in which we maniplulate subsets of some fixed set X often arises, and it is useful to introduce a special type of relative complement with respect to X. For any subset A of X, the *complement* $\overline{A}$ of A in X is defined by

$$\overline{A} = X - A,$$

which can also be expressed as

$$\overline{A} = \{x \in X \mid x \notin A\}.$$

Using the union operation, we can form bigger sets by taking unions with singletons. For example, we can form

$$\{a, b, c\} = \{a, b\} \cup \{c\}.$$

Remark: We can systematically construct bigger and bigger sets by the following method: given any set A let

$$A^+ = A \cup \{A\}.$$

If we start from the empty set, we obtain the sets

$$\emptyset,\ \{\emptyset\},\ \ \{\emptyset, \{\emptyset\}\},\ \ \{\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}\},\ \ etc.$$

These sets can be used to define the natural numbers and the $+$ operation corresponds to the successor function on the natural numbers (i.e., $n \mapsto n+1$).

The algebraic properties of union, intersection, and complementation are inherited from the properties of disjunction, conjunction, and negation. The following proposition lists some of the most important properties of union, intersection, and complementation. Some of these properties are versions of Proposition 1.2 for subsets.

Proposition 1.8. *The following equations hold for all sets A, B, C:*

$$\begin{aligned} A \cup \emptyset &= A \\ A \cap \emptyset &= \emptyset \\ A \cup A &= A \\ A \cap A &= A \\ A \cup B &= B \cup A \\ A \cap B &= B \cap A. \end{aligned}$$

The last two assert the commutativity of $\cup$ and $\cap$. We have distributivity of $\cap$ over $\cup$ and of $\cup$ over $\cap$:

$$\begin{aligned} A \cap (B \cup C) &= (A \cap B) \cup (A \cap C) \\ A \cup (B \cap C) &= (A \cup B) \cap (A \cup C). \end{aligned}$$

We have associativity of $\cap$ and $\cup$:

$$\begin{aligned} A \cap (B \cap C) &= (A \cap B) \cap C \\ A \cup (B \cup C) &= (A \cup B) \cup C. \end{aligned}$$

Proof. We use Proposition 1.2. Let us prove that $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$, leaving the proof of the other equations as an exercise. We prove the two inclusions $A \cap (B \cup C) \subseteq (A \cap B) \cup (A \cap C)$ and $(A \cap B) \cup (A \cap C) \subseteq A \cap (B \cup C)$.

Assume that $x \in A \cap (B \cup C)$. This means that $x \in A$ and $x \in B \cup C$; that is,

$$(x \in A) \wedge ((x \in B) \vee (x \in C)).$$

Using the distributivity of $\wedge$ over $\vee$, we obtain

$$((x \in A) \wedge (x \in B)) \vee ((x \in A) \wedge (x \in C)).$$

But the above says that $x \in (A \cap B) \cup (A \cap C)$, which proves our first inclusion.

Conversely assume that $x \in (A \cap B) \cup (A \cap C)$. This means that $x \in (A \cap B)$ or $x \in (A \cap C)$; that is,

$$((x \in A) \wedge (x \in B)) \vee ((x \in A) \wedge (x \in C)).$$

Using the distributivity of $\wedge$ over $\vee$ (in the other direction), we obtain

$$(x \in A) \wedge ((x \in B) \vee (x \in C)),$$

which says that $x \in A \cap (B \cup C)$, and proves our second inclusion.

Note that we could have avoided two arguments by proving that $x \in A \cap (B \cup C)$ iff $(A \cap B) \cup (A \cap C)$ using the fact that the distributivity of $\wedge$ over $\vee$ is a logical equivalence. $\square$

We also have the following version of Proposition 1.1 for subsets.

Proposition 1.9. *For every set X and any two subsets A, B of X, the following identities hold:*

$$\begin{aligned}\overline{\overline{A}} &= A \\ \overline{(A \cap B)} &= \overline{A} \cup \overline{B} \\ \overline{(A \cup B)} &= \overline{A} \cap \overline{B}.\end{aligned}$$

The last two are de Morgan laws.

Another operation is the power set formation. It is indeed a "powerful" operation, in the sense that it allows us to form very big sets.

Definition 1.7. Given any set A, there is a set $\mathcal{P}(A)$ also denoted 2^A called the *power set of A* whose members are exactly the subsets of A; that is,

$$X \in \mathcal{P}(A) \quad \text{iff} \quad X \subseteq A.$$

For example, if $A = \{a, b, c\}$, then

$$\mathcal{P}(A) = \{\emptyset, \{a\}, \{b\}, \{c\}, \{a, b\}, \{a, c\}, \{b, c\}, \{a, b, c\}\},$$

a set containing eight elements. Note that the empty set and A itself are always members of $\mathcal{P}(A)$.

Remark: If A has n elements, it is not hard to show that $\mathcal{P}(A)$ has 2^n elements. For this reason, many people, including me, prefer the notation 2^A for the power set of A.

It is possible to define the union of possibly infinitely many sets. Given any set X (think of X as a set of sets), there is a set $\bigcup X$ defined so that

$$x \in \bigcup X \quad \text{iff} \quad \exists B (B \in X \wedge x \in B).$$

This says that $\bigcup X$ consists of all elements that belong to some member of X.

If we take $X = \{A, B\}$, where A and B are two sets, we see that

$$\bigcup\{A, B\} = A \cup B.$$

Observe that

$$\bigcup\{A\} = A, \quad \bigcup\{A_1, \ldots, A_n\} = A_1 \cup \cdots \cup A_n.$$

and in particular, $\bigcup \emptyset = \emptyset$.

We can also define infinite intersections. For every nonempty set X there is a set $\bigcap X$ defined by

$$x \in \bigcap X \quad \text{iff} \quad \forall B(B \in X \Rightarrow x \in B).$$

Observe that

$$\bigcap\{A, B\} = A \cap B, \quad \bigcap\{A_1, \ldots, A_n\} = A_1 \cap \cdots \cap A_n.$$

However, $\bigcap \emptyset$ is undefined. Indeed, $\bigcap \emptyset$ would have to be the set of all sets, since the condition

$$\forall B(B \in \emptyset \Rightarrow x \in B)$$

holds trivially for all B (as the empty set has no members). However there is no such set, because its existence would lead to a paradox! This point is discussed is Chapter 2. Let us simply say that dealing with big infinite sets is tricky.

Thorough and yet accessible presentations of set theory can be found in Halmos [27] and Enderton [11].

We close this chapter with a quick discussion of induction on the natural numbers.

1.12 Induction and The Well–Ordering Principle on the Natural Numbers

Recall that the set of natural numbers is the set $\mathbb{N}$ given by

$$\mathbb{N} = \{0, 1, 2, 3, \ldots\}.$$

In this chapter we do not attempt to define the natural numbers from other concepts, such as sets. We assume that they are "God given." One of our main goals is to prove properties of the natural numbers. For this, certain subsets called inductive play a crucial role.

Definition 1.8. We say that a subset S of $\mathbb{N}$ is *inductive* iff

(1) $0 \in S$.

(2) For every $n \in S$, we have $n + 1 \in S$.

One of the most important proof principles for the natural numbers is the following:

Proof Template 1.19. *(Induction Principle for $\mathbb{N}$)*
Every inductive subset S of $\mathbb{N}$ is equal to $\mathbb{N}$ itself; that is $S = \mathbb{N}$.

Let us give one example illustrating Proof Template 1.19.

Example 1.23. We prove that for every real number $a \neq 1$ and evey natural number n, we have

$$1 + a + \cdots + a^n = \frac{a^{n+1} - 1}{a - 1}.$$

This can also be written as

$$\sum_{i=1}^{n} a^i = \frac{a^{n+1} - 1}{a - 1}, \qquad (*)$$

with the convention that $a^0 = 1$, even if $a = 0$. Let S be the set of natural numbers n for which the identity $(*)$ holds, and let us prove that S is inductive.

First we need to prove that $0 \in S$. The lefthand side becomes $a^0 = 1$, and the righthand side is $(a-1)/(a-1)$, which is equal to 1 since we assume that $a \neq 1$. Therefore, $(*)$ holds for $n = 0$; that is, $0 \in S$.

Next assume that $n \in S$ (this is called the *induction hypothesis*). We need to prove that $n + 1 \in S$. Observe that

$$\sum_{i=1}^{n+1} a^i = \sum_{i=1}^{n} a^i + a^{n+1}.$$

Now since we assumed that $n \in S$, we have

$$\sum_{i=1}^{n} a^i = \frac{a^{n+1} - 1}{a - 1},$$

and we deduce that

$$\begin{aligned} \sum_{i=1}^{n+1} a^i &= \sum_{i=1}^{n} a^i + a^{n+1} \\ &= \frac{a^{n+1} - 1}{a - 1} + a^{n+1} \\ &= \frac{a^{n+1} - 1 + a^{n+2} - a^{n+1}}{a - 1} \\ &= \frac{a^{n+2} - 1}{a - 1}. \end{aligned}$$

This proves that $n + 1 \in S$. Therefore, S is inductive, and so $S = \mathbb{N}$.

Another important property of $\mathbb{N}$ is the so-called *well–ordering principle*. *This principle turns out to be equivalent to the induction principle for* $\mathbb{N}$. In this chapter we accept the well–ordering principle without proof.

Proof Template 1.20. *(Well–Ordering Principle for* $\mathbb{N}$*)*
Every nonempty subset of $\mathbb{N}$ *has a smallest element.*

Proof Template 1.20 can be used to prove properties of $\mathbb{N}$ by contradiction. For example, consider the property that every natural number n is either even or odd.

For the sake of contradiction (here, we use the proof–by–contradiction principle), assume that our statement does not hold. If so, the subset S of natural numbers n for which n is neither even nor odd is nonempty. By the well–ordering principle, the set S has a smallest element, say m.

If $m = 0$, then 0 would be neither even nor odd, a contradiction since 0 is even. Therefore, $m > 0$. But then, $m - 1 \notin S$, since m is the smallest element of S. This means that $m - 1$ is either even or odd. But if $m - 1$ is even, then $m - 1 = 2k$ for some k, so $m = 2k + 1$ is odd, and if $m - 1$ is odd, then $m - 1 = 2k + 1$ for some k, so $m = 2(k + 1)$ is even. We just proved that m is either even or odd, contradicting the fact that $m \in S$. Therefore, S must be empty and we proved the desired result.

We conclude this section with one more example showing the usefulness of the well–ordering principle.

Example 1.24. Suppose we have a property $P(n)$ of the natural numbers such that $P(n)$ holds for at least some n, and that for every n such that $P(n)$ holds and $n \geq 100$, then there is some $m < n$ such that $P(m)$ holds. We claim that there is some $m < 100$ such that $P(m)$ holds. Let S be the set of natural numbers n such that $P(n)$ holds. By hypothesis, there is some some n such that $P(n)$ holds, so S is nonempty. By the well–ordering principle, the set S has a smallest element, say m. For the sake of contradiction, assume that $m \geq 100$. Then since $P(m)$ holds and $m \geq 100$, by the hypothesis there is some $m' < m$ such that $P(m')$ holds, contradicting the fact that m is the smallest element of S. Therefore, by the proof–by–contradiction principle, we conclude that $m < 100$, as claimed.

Beware that the well–ordering principle is false for $\mathbb{Z}$, because $\mathbb{Z}$ does not have a smallest element.

1.13 Summary

The main goal of this chapter is to describe how to construct proofs in terms of *proof templates*. A brief and informal introduction to sets and set operations is also provided.

- We describe the syntax of *propositions*.
- We define the proof templates for *implication*.
- We show that *deductions* proceed from *assumptions* (or *premises*) according to *proof templates*.

- We introduce falsity $\perp$ and negation $\neg P$ as an abbrevation for $P \Rightarrow \perp$. We describe the proof templates for conjunction, disjunction, and negation.
- We show that one of the rules for negation is the *proof–by–contradiction* rule (also known as *RAA*). It plays a special role, in the sense that it allows for the construction of indirect proofs.
- We present the *proof–by–contrapositive rule*.
- We present the *de Morgan laws* as well as some basic properties of $\vee$ and $\wedge$.
- We give some examples of proofs of "real" statements.
- We give an example of a nonconstructive proof of the statement: there are two irrational numbers, a and b, so that a^b is rational.
- We explain the *truth-value semantics* of propositional logic.
- We define the *truth tables* for the boolean functions associated with the logical connectives (and, or, not, implication, equivalence).
- We define the notion of *validity* and *tautology*.
- We discuss *soundness* (or *consistency*) and *completeness*.
- We state the *soundness and completeness theorems* for propositional classical logic.
- We explain how to use *counterexamples* to prove that certain propositions are not provable.
- We add *first-order quantifiers* ("for all" $\forall$ and "there exists" $\exists$) to the language of propositional logic and define *first-order logic*.
- We describe *free* and *bound* variables.
- We describe Proof Templates for the quantifiers.
- We prove some "de Morgan"-type rules for the quantified formulae.
- We introduce *sets* and explain when two sets are equal.
- We define the notion of *subset*.
- We define some basic operations on sets: the *union* $A \cup B$, *intersection* $A \cap B$, and *relative complement* $A - B$.
- We define the *complement* of a subset of a given set.

- We prove some basic properties of union, intersection and complementation, including the *de Morgan laws*.
- We define the *power set* of a set.
- We define *inductive subsets* of $\mathbb{N}$ and state the *induction principle for* $\mathbb{N}$.
- We state the *well–ordering principle* for $\mathbb{N}$.

Problems

Problem 1.1. Give a proof of the proposition $(P \Rightarrow Q) \Rightarrow ((P \Rightarrow (Q \Rightarrow R)) \Rightarrow (P \Rightarrow R))$.

Problem 1.2. (a) Prove the "de Morgan" laws:

$$\begin{aligned}\neg(P \wedge Q) &\equiv \neg P \vee \neg Q \\ \neg(P \vee Q) &\equiv \neg P \wedge \neg Q.\end{aligned}$$

(b) Prove the propositions $(P \wedge \neg Q) \Rightarrow \neg(P \Rightarrow Q)$ and $\neg(P \Rightarrow Q) \Rightarrow (P \wedge \neg Q)$.

Problem 1.3. (a) Prove the equivalences

$$\begin{aligned}P \vee P &\equiv P \\ P \wedge P &\equiv P \\ P \vee Q &\equiv Q \vee P \\ P \wedge Q &\equiv Q \wedge P.\end{aligned}$$

(b) Prove the equivalences

$$\begin{aligned}P \wedge (P \vee Q) &\equiv P \\ P \vee (P \wedge Q) &\equiv P.\end{aligned}$$

Problem 1.4. Prove the propositions

$$\begin{aligned}&P \Rightarrow (Q \Rightarrow (P \wedge Q)) \\ &(P \Rightarrow Q) \Rightarrow ((P \Rightarrow \neg Q) \Rightarrow \neg P) \\ &(P \Rightarrow R) \Rightarrow ((Q \Rightarrow R) \Rightarrow ((P \vee Q) \Rightarrow R)).\end{aligned}$$

Problem 1.5. Prove the following equivalences:

$$\begin{aligned}P \wedge (P \Rightarrow Q) &\equiv P \wedge Q \\ Q \wedge (P \Rightarrow Q) &\equiv Q \\ \big(P \Rightarrow (Q \wedge R)\big) &\equiv \big((P \Rightarrow Q) \wedge (P \Rightarrow R)\big).\end{aligned}$$

Problem 1.6. Prove the propositions

$$
\begin{aligned}
&(P \Rightarrow Q) \Rightarrow \neg\neg(\neg P \vee Q)\\
&\neg\neg(\neg\neg P \Rightarrow P).
\end{aligned}
$$

Problem 1.7. Prove the proposition $\neg\neg(P \vee \neg P)$.

Problem 1.8. Prove the propositions

$$(P \vee \neg P) \Rightarrow (\neg\neg P \Rightarrow P) \quad \text{and} \quad (\neg\neg P \Rightarrow P) \Rightarrow (P \vee \neg P).$$

Problem 1.9. Prove the propositions

$$(P \Rightarrow Q) \Rightarrow \neg\neg(\neg P \vee Q) \quad \text{and} \quad (\neg P \Rightarrow Q) \Rightarrow \neg\neg(P \vee Q).$$

Problem 1.10. (a) Prove the distributivity of $\wedge$ over $\vee$ and of $\vee$ over $\wedge$:

$$
\begin{aligned}
P \wedge (Q \vee R) &\equiv (P \wedge Q) \vee (P \wedge R)\\
P \vee (Q \wedge R) &\equiv (P \vee Q) \wedge (P \vee R).
\end{aligned}
$$

(b) Prove the associativity of $\wedge$ and $\vee$:

$$
\begin{aligned}
P \wedge (Q \wedge R) &\equiv (P \wedge Q) \wedge R\\
P \vee (Q \vee R) &\equiv (P \vee Q) \vee R.
\end{aligned}
$$

Problem 1.11. (a) Let $X = \{X_i \mid 1 \leq i \leq n\}$ be a finite family of sets. Prove that if $X_{i+1} \subseteq X_i$ for all i, with $1 \leq i \leq n-1$, then

$$\bigcap X = X_n.$$

Prove that if $X_i \subseteq X_{i+1}$ for all i, with $1 \leq i \leq n-1$, then

$$\bigcup X = X_n.$$

(b) Recall that $\mathbb{N}_+ = \mathbb{N} - \{0\} = \{1, 2, 3, \ldots, n, \ldots\}$. Give an example of an infinite family of sets, $X = \{X_i \mid i \in \mathbb{N}_+\}$, such that

1. $X_{i+1} \subseteq X_i$ for all $i \geq 1$.
2. X_i is infinite for every $i \geq 1$.
3. $\bigcap X$ has a single element.

(c) Give an example of an infinite family of sets, $X = \{X_i \mid i \in \mathbb{N}_+\}$, such that

1. $X_{i+1} \subseteq X_i$ for all $i \geq 1$.

2. X_i is infinite for every $i \geq 1$.

3. $\bigcap X = \emptyset$.

Problem 1.12. An integer, $n \in \mathbb{Z}$, is divisible by 3 iff $n = 3k$, for some $k \in \mathbb{Z}$. Thus (by the division theorem), an integer, $n \in \mathbb{Z}$, is not divisible by 3 iff it is of the form $n = 3k+1, 3k+2$, for some $k \in \mathbb{Z}$ (you don't have to prove this).

Prove that for any integer, $n \in \mathbb{Z}$, if n^2 is divisible by 3, then n is divisible by 3.

Hint. Prove the contrapositive. If n of the form $n = 3k + 1, 3k + 2$, then so is n^2 (for a different k).

Problem 1.13. Use Problem 1.12 to prove that $\sqrt{3}$ is irrational, that is, $\sqrt{3}$ can't be written as $\sqrt{3} = p/q$, with $p, q \in \mathbb{Z}$ and $q \neq 0$.

Problem 1.14. Prove that $b = \log_2 9$ is irrational. Then, prove that $a = \sqrt{2}$ and $b = \log_2 9$ are two irrational numbers such that a^b is rational.

Chapter 2

Mathematical Reasoning And Logic, A Deeper View

2.1 Introduction

This chapter is a more advanced and more formal version of Chapter 1. The reader should review Chapter 1 before reading this chapter which relies rather heavily on it.

As in Chapter 1 , the goal of this chapter is to provide an answer to the question, "What is a proof?" We do so by formalizing the basic rules of reasoning that we use, most of the time subconsciously, in a certain kind of formalism known as a *natural deduction system*. We give a (very) quick introduction to *mathematical logic*, with a very deliberate *proof-theoretic* bent, that is, neglecting almost completely all semantic notions, except at a very intuitive level. We still feel that this approach is fruitful because the mechanical and rules-of-the-game flavor of proof systems is much more easily grasped than semantic concepts. In this approach, we follow Peter Andrews' motto [1]:

"To truth through proof."

We present various natural deduction systems due to Prawitz and Gentzen (in more modern notation), both in their intuitionistic and classical version. The adoption of natural deduction systems as proof systems makes it easy to question the validity of some of the inference rules, such as the *principle of proof by contradiction*. In brief, we try to explain to our readers the difference between *constructive* and *classical* (i.e., not necessarily constructive) proofs. In this respect, we plant the seed that there is a deep relationship between *constructive proofs* and the notion of *computation* (the "Curry–Howard isomorphism" or "formulae-as-types principle," see Section 2.12 and Howard [30]).

2.2 Inference Rules, Deductions, The Proof Systems $\mathcal{N}_m^{\Rightarrow}$ and $\mathcal{NG}_m^{\Rightarrow}$

In this section we review some basic proof principles and attempt to clarify, at least informally, what constitutes a mathematical proof.

In order to define the notion of proof rigorously, we would have to define a formal language in which to express statements very precisely and we would have to set up a proof system in terms of axioms and proof rules (also called inference rules). We do not go into this as this would take too much time. Instead, we content ourselves with an intuitive idea of what a statement is and focus on stating as precisely as possible the rules of logic that are used in constructing proofs. Readers who really want to see a thorough (and rigorous) introduction to logic are referred to Gallier [19], van Dalen [51], or Huth and Ryan [31], a nice text with a computer science flavor. A beautiful exposition of logic (from a proof-theoretic point of view) is also given in Troelstra and Schwichtenberg [50], but at a more advanced level. Frank Pfenning has also written an excellent and more extensive introduction to constructive logic. This is available on the web at

http://www.andrew.cmu.edu/course/15-317/handouts/logic.pdf

We also highly recommend the beautifully written little book by Timothy Gowers (Fields Medalist, 1998) [26] which, among other things, discusses the notion of proof in mathematics (as well as the necessity of formalizing proofs without going overboard).

In mathematics and computer science, we **prove statements.** Recall that statements may be *atomic* or *compound*, that is, built up from simpler statements using *logical connectives*, such as *implication* (if–then), *conjunction* (and), *disjunction* (or), *negation* (not), and (existential or universal) *quantifiers*.

As examples of atomic statements, we have:

1. "A student is eager to learn."
2. "The product of two odd integers is odd."

Atomic statements may also contain "variables" (standing for arbitrary objects). For example

1. human(x): "x is a human."
2. needs-to-drink(x): "x needs to drink."

An example of a compound statement is

$$\text{human}(x) \Rightarrow \text{needs-to-drink}(x).$$

In the above statement, $\Rightarrow$ is the symbol used for logical implication. If we want to assert that every human needs to drink, we can write

$$\forall x(\text{human}(x) \Rightarrow \text{needs-to-drink}(x));$$

this is read: "For every x, if x is a human then x needs to drink."

If we want to assert that some human needs to drink we write

$$\exists x(\text{human}(x) \Rightarrow \text{needs-to-drink}(x));$$

this is read: "There is some x such that, if x is a human then x needs to drink."

We often denote statements (also called *propositions* or *(logical) formulae*) using letters, such as A, B, P, Q, and so on, typically upper-case letters (but sometimes Greek letters, φ, ψ, etc.).

Recall from Section 1.2 that *Compound statements* are defined as follows: If P and Q are statements, then

1. the *conjunction* of P and Q is denoted $P \wedge Q$ (pronounced, P and Q),
2. the *disjunction* of P and Q is denoted $P \vee Q$ (pronounced, P or Q),
3. the *implication* of P and Q is denoted by $P \Rightarrow Q$ (pronounced, if P then Q, or P implies Q).

Instead of using the symbol $\Rightarrow$, some authors use the symbol $\to$ and write an implication as $P \to Q$. We do not like to use this notation because the symbol $\to$ is already used in the notation for functions ($f\colon A \to B$). The symbol $\supset$ is sometimes used instead of $\Rightarrow$. We mostly use the symbol $\Rightarrow$.

We also have the atomic statements $\perp$ (*falsity*), think of it as the statement that is false no matter what; and the atomic statement $\top$ (*truth*), think of it as the statement that is always true.

The constant $\perp$ is also called *falsum* or *absurdum*. It is a formalization of the notion of *absurdity inconsistency* (a state in which contradictory facts hold).

Given any proposition P it is convenient to define

4. the *negation* $\neg P$ of P (pronounced, not P) as $P \Rightarrow \perp$. Thus, $\neg P$ (sometimes denoted $\sim P$) is just a shorthand for $P \Rightarrow \perp$. We write $\neg P \equiv (P \Rightarrow \perp)$.

The intuitive idea is that $\neg P \equiv (P \Rightarrow \perp)$ is true if and only if P is false. Actually, because we don't know what truth is, it is "safer" (and more constructive) to say that $\neg P$ is provable if and only if for every proof of P we can derive a contradiction (namely, $\perp$ is provable). In particular, P should not be provable. For example, $\neg(Q \wedge \neg Q)$ is provable (as we show later, because any proof of $Q \wedge \neg Q$ yields a proof of $\perp$). However, the fact that a proposition P is **not** provable does not imply that $\neg P$ **is** provable. There are plenty of propositions such that both P and $\neg P$ are not provable, such as $Q \Rightarrow R$, where Q and R are two unrelated propositions (with no common symbols).

Whenever necessary to avoid ambiguities, we add matching parentheses: $(P \wedge Q)$, $(P \vee Q)$, $(P \Rightarrow Q)$. For example, $P \vee Q \wedge R$ is ambiguous; it means either $(P \vee (Q \wedge R))$ or $((P \vee Q) \wedge R)$.

Another important logical operator is *equivalence*.

If P and Q are statements, then

5. the *equivalence* of P and Q is denoted $P \equiv Q$ (or $P \Longleftrightarrow Q$); it is an abbreviation for $(P \Rightarrow Q) \land (Q \Rightarrow P)$. We often say "$P$ if and only if Q" or even "P iff Q" for $P \equiv Q$.

To prove a logical equivalence $P \equiv Q$, we have to prove **both** implications $P \Rightarrow Q$ and $Q \Rightarrow P$.

As discussed in Sections 1.2 and 1.3, the meaning of the logical connectives $(\land, \lor, \Rightarrow, \neg, \equiv)$ is intuitively clear. This is certainly the case for *and* $(\land)$, since a conjunction $P \land Q$ is true if and only if both P and Q are true (if we are not sure what "true" means, replace it by the word "provable"). However, for *or* $(\lor)$, do we mean inclusive or or exclusive or? In the first case, $P \lor Q$ is true if both P and Q are true, but in the second case, $P \lor Q$ is false if both P and Q are true (again, in doubt change "true" to "provable"). We always mean inclusive or. The situation is worse for *implication* $(\Rightarrow)$. When do we consider that $P \Rightarrow Q$ is true (provable)? The answer is that it depends on the rules! The "classical" answer is that $P \Rightarrow Q$ is false (not provable) if and only if P is true and Q is false.

Of course, there are problems with the above paragraph. What does truth have to do with all this? What do we mean when we say, "P is true"? What is the relationship between truth and provability?

These are actually deep (and tricky) questions whose answers are not so obvious. One of the major roles of logic is to clarify the notion of truth and its relationship to provability. We avoid these fundamental issues by dealing exclusively with the notion of proof. So, the big question is: what is a proof?

An alternative view (that of intuitionistic logic) of the meaning of implication is that any proof of $P \Rightarrow Q$ can be used to construct a proof of Q given any proof of P. As a consequence of this interpretation, we show later that if $\neg P$ is provable, then $P \Rightarrow Q$ is also provable (instantly) whether or not Q is provable. In such a situation, we often say that $P \Rightarrow Q$ is *vacuously provable*.

2.3 Proof Rules, Deduction and Proof Trees for Implication

During the process of constructing a proof, it may be necessary to introduce a list of *hypotheses*, also called *premises* (or *assumptions*), which grows and shrinks during the proof. When a proof is finished, *it should have an empty list of premises*. As we show shortly, this amounts to proving implications of the form

$$(P_1 \land P_2 \land \cdots \land P_n) \Rightarrow Q.$$

However, there are certain advantages in defining the notion of *proof* (or *deduction*) of a proposition from a set of premises. Sets of premises are usually denoted using upper-case Greek letters such as Γ or Δ.

Roughly speaking, a *deduction* of a proposition Q from a multiset of premises Γ is a finite labeled tree whose root is labeled with Q (the *conclusion*), whose leaves are labeled

Figure 2.1: David Hilbert, 1862–1943 (left and middle), Gerhard Gentzen, 1909–1945 (middle right), and Dag Prawitz, 1936– (right)

with premises from Γ (possibly with multiple occurrences), and such that every interior node corresponds to a given set of *proof rules* (or *inference rules*). In Chapter 1, proof rules were called proof templates. Certain simple deduction trees are declared as obvious proofs, also called *axioms*. The process of managing the list of premises during a proof is a bit technical and can be achieved in various ways. We will present a method due to Prawitz and another method due to Gentzen.

There are many kinds of proof systems: Hilbert-style systems, natural-deduction systems, Gentzen sequents systems, and so on. We describe a so-called *natural deduction system* invented by G. Gentzen in the early 1930s (and thoroughly investigated by D. Prawitz in the mid 1960s).

The major advantage of this system is that *it captures quite nicely the "natural" rules of reasoning that one uses when proving mathematical statements*. This does not mean that it is easy to find proofs in such a system or that this system is indeed very intuitive. We begin with the inference rules for implication and first consider the following question.

How do we proceed to prove an implication, $A \Rightarrow B$? The proof rule corresponds to Proof Template 1.2 (Implication–Intro) and the reader may want to first review the examples discussed in Section 1.3. The rule, called $\Rightarrow$*-intro*, is: *assume that A has already been proven and then prove B, making as many uses of A as needed*.

An important point is that a proof should not depend on any "open" assumptions and to address this problem we introduce a mechanism of "discharging" or "closing" premises, as we discussed in Section 1.3.

What this means is that certain rules of our logic are required to discard (the usual terminology is "discharge") certain occurrences of premises so that the resulting proof does not depend on these premises.

Technically, there are various ways of implementing the discharging mechanism but they all involve some form of tagging (with a "new" variable). For example, the rule formalizing the process that we have just described to prove an implication, $A \Rightarrow B$, known as $\Rightarrow$-*introduction*, uses a tagging mechanism described precisely in Definition 2.1.

Now, the rule that we have just described is not sufficient to prove certain propositions that should be considered provable under the "standard" intuitive meaning of implication. For example, after a moment of thought, I think most people would want the proposition

$P \Rightarrow ((P \Rightarrow Q) \Rightarrow Q)$ to be provable. If we follow the procedure that we have advocated, we assume both P and $P \Rightarrow Q$ and we try to prove Q. For this, we need a new rule, namely:

If P and $P \Rightarrow Q$ are both provable, then Q is provable.

The above rule is known as the *$\Rightarrow$-elimination rule* (or *modus ponens*) and it is formalized in tree-form in Definition 2.1. It corresponds to Proof Template 1.3.

We now make the above rules precise and for this, we represent proofs and deductions as certain kinds of trees and *view the logical rules (inference rules) as tree-building rules*. In the definition below, the expression Γ, P stands for the multiset obtained by adding one more occurrence of P to Γ. So, P *may already belong* to Γ. Similarly, if Γ and Δ are two multisets of propositions, then Γ, Δ denotes the union of Γ and Δ *as a multiset*, which means that if P occurs k_1 times in Γ and P occurs k_2 times in Δ, then P occurs $k_1 + k_2$ times in Γ, Δ ($k_1, k_2 \in \mathbb{N}$).

A picture such as

$$\begin{array}{c} \Delta \\ \mathcal{D} \\ P \end{array}$$

represents a deduction tree $\mathcal{D}$ whose root is labeled with P and whose leaves are labeled with propositions from the *multiset* Δ (a set possibly with multiple occurrences of its members). Some of the propositions in Δ may be tagged by variables. The list of untagged propositions in Δ is the list of *premises* of the deduction tree. We often use an abbreviated version of the above notation where we omit the deduction $\mathcal{D}$, and simply write

$$\begin{array}{c} \Delta \\ P. \end{array}$$

For example, in the deduction tree below,

$$\cfrac{\cfrac{P \Rightarrow (R \Rightarrow S) \qquad P}{R \Rightarrow S} \qquad \cfrac{Q \Rightarrow R \qquad \cfrac{P \Rightarrow Q \qquad P}{Q}}{R}}{S}$$

no leaf is tagged, so the premises form the multiset

$$\Delta = \{P \Rightarrow (R \Rightarrow S), P, Q \Rightarrow R, P \Rightarrow Q, P\},$$

with two occurrences of P, and the conclusion is S.

As we saw in our earlier example, certain inferences rules have the effect that some of the original premises may be discarded; the traditional jargon is that some premises may be *discharged* (or *closed*). This is the case for the inference rule whose conclusion is an implication. When one or several occurrences of some proposition P are discharged by an inference rule, these occurrences (which label some leaves) are tagged with some new variable not already appearing in the deduction tree. If x is a new tag, the tagged occurrences of P

are denoted P^x and we indicate the fact that premises were discharged by that inference by writing x immediately to the right of the inference bar. For example,

$$\dfrac{\dfrac{P^x, Q}{Q}}{P \Rightarrow Q}\; x$$

is a deduction tree in which the premise P is discharged by the inference rule. This deduction tree only has Q as a premise, inasmuch as P is discharged.

What is the meaning of the horizontal bars? Actually, nothing really. Here, we are victims of an old habit in logic. Observe that there is always a single proposition immediately under a bar but there may be several propositions immediately above a bar. The intended meaning of the bar is that the proposition below it is obtained as the result of applying an inference rule to the propositions above it. For example, in

$$\dfrac{Q \Rightarrow R \qquad Q}{R}$$

the proposition R is the result of applying the $\Rightarrow$-elimination rule (see Definition 2.1 below) to the two premises $Q \Rightarrow R$ and Q. Thus, the use of the bar is just a convention used by logicians going back at least to the 1900s. Removing the bar everywhere would not change anything in our trees, except perhaps reduce their readability. Most logic books draw proof trees using bars to indicate inferences, therefore we also use bars in depicting our proof trees.

Because propositions do not arise from the vacuum but instead are built up from a set of atomic propositions using logical connectives (here, $\Rightarrow$), we assume the existence of an "official set of atomic propositions," or set of *propositional symbols*, $\mathbf{PS} = \{\mathbf{P}_1, \mathbf{P}_2, \mathbf{P}_3, \ldots\}$. So, for example, $\mathbf{P}_1 \Rightarrow \mathbf{P}_2$ and $\mathbf{P}_1 \Rightarrow (\mathbf{P}_2 \Rightarrow \mathbf{P}_1)$ are propositions. Typically, we use upper-case letters such as P, Q, R, S, A, B, C, and so on, to denote arbitrary propositions formed using atoms from $\mathbf{PS}$.

Definition 2.1. The axioms, inference rules, and deduction trees for *implicational logic* are defined as follows.

Axioms.

(i) Every one-node tree labeled with a single proposition P is a deduction tree for P with set of premises $\{P\}$.

(ii) The tree

$$\dfrac{\Gamma, P}{P}$$

is a deduction tree for P with multiset set of premises Γ, P.

The above is a concise way of denoting a two-node tree with its leaf labeled with the multiset consisting of P and the propositions in Γ, each of these propositions (including P) having possibly multiple occurrences but at least one, and whose root is labeled with P. A more explicit form is

$$\frac{\overbrace{P_1, \cdots, P_1}^{k_1}, \cdots, \overbrace{P_i, \cdots, P_i}^{k_i}, \cdots, \overbrace{P_n, \cdots, P_n}^{k_n}}{P_i},$$

where $k_1, \ldots, k_n \geq 1$ and $n \geq 1$. This axiom says that we always have a deduction of P_i from any set of premises including P_i. They correspond to the Proof Template 1.1 (Trivial Deduction).

The $\Rightarrow$**-introduction rule**.

If $\mathcal{D}$ is a deduction tree for Q from the premises in Γ and one or more occurrences of the proposition P, then

$$\frac{\begin{array}{c}\Gamma, P^x \\ \mathcal{D} \\ Q\end{array}}{P \Rightarrow Q}\ x$$

is a deduction tree for $P \Rightarrow Q$ from Γ.

This proof rule is a formalization of Proof Template 1.2 (Implication–Intro). Note that this inference rule has the additional effect of discharging a nonempty set of occurrences of the premise P (which label leaves of the deduction $\mathcal{D}$). These occurrences are tagged with a new variable x, and the tag x is also placed immediately to the right of the inference bar. This is a reminder that the deduction tree whose conclusion is $P \Rightarrow Q$ *no longer has the occurrences of* P *labeled with* x *as premises*.

The $\Rightarrow$**-elimination rule**.

If $\mathcal{D}_1$ is a deduction tree for $P \Rightarrow Q$ from the premises Γ and $\mathcal{D}_2$ is a deduction for P from the premises Δ, then

$$\frac{\begin{array}{cc}\Gamma & \Delta \\ \mathcal{D}_1 & \mathcal{D}_2 \\ P \Rightarrow Q & P\end{array}}{Q}$$

is a deduction tree for Q from the premises in the multiset Γ, Δ. This rule is also known as *modus ponens*. This proof rule is a formalization of Proof Template 1.3 (Implication–Elim).

In the above axioms and rules, Γ or Δ may be empty; P, Q denote arbitrary propositions built up from the atoms in **PS**; and $\mathcal{D}, \mathcal{D}_1$, and $\mathcal{D}_2$ denote deductions, possibly a one-node tree.

A *deduction tree* is either a one-node tree labeled with a single proposition or a tree constructed using the above axioms and rules. A *proof tree* is a deduction tree such that *all its premises are discharged*. The above proof system is denoted $\mathcal{N}_m^{\Rightarrow}$ (here, the subscript m stands for *minimal*, referring to the fact that this a bare-bones logical system).

Observe that a proof tree has at least two nodes. A proof tree Π for a proposition P may be denoted

$$\begin{array}{c}\Pi \\ P\end{array}$$

with an empty set of premises (we don't display $\emptyset$ on top of Π). We tend to denote deductions by the letter $\mathcal{D}$ and proof trees by the letter Π, possibly subscripted.

We emphasize that the $\Rightarrow$-introduction rule says that in order to prove an implication $P \Rightarrow Q$ from a set of premises Γ, we assume that P has already been proven, add P to the premises in Γ, and then prove Q from Γ and P. Once this is done, *the premise P is deleted*.

This rule formalizes the kind of reasoning that we all perform whenever we prove an implication statement. In that sense, it is a natural and familiar rule, except that we perhaps never stopped to think about what we are really doing. However, the business about discharging the premise P when we are through with our argument is a bit puzzling. Most people probably never carry out this "discharge step" consciously, but such a process does take place implicitly.

Remarks:

1. Only the leaves of a deduction tree may be discharged. Interior nodes, including the root, are *never* discharged.

2. Once a set of leaves labeled with some premise P marked with the label x has been discharged, none of these leaves can be discharged again. So, each label (say x) *can only be used once*. This corresponds to the fact that some leaves of our deduction trees get "killed off" (discharged).

3. A proof is a deduction tree whose leaves are *all discharged* (Γ is empty). This corresponds to the philosophy that if a proposition has been proven, then the validity of the proof should not depend on any assumptions that are still active. We may think of a deduction tree as an unfinished proof tree.

4. When constructing a proof tree, we have to be careful not to include (accidentally) extra premises that end up not being discharged. If this happens, we probably made a mistake and the redundant premises should be deleted. On the other hand, if we have a proof tree, we can always add extra premises to the leaves and create a new proof tree from the previous one by discharging all the new premises.

5. Beware, when we deduce that an implication $P \Rightarrow Q$ is provable, we **do not** prove that P **and** Q are provable; we only prove that **if** P is provable, **then** Q is provable.

The $\Rightarrow$-elimination rule formalizes the use of *auxiliary lemmas*, a mechanism that we use all the time in making mathematical proofs. Think of $P \Rightarrow Q$ as a lemma that has already been established and belongs to some database of (useful) lemmas. This lemma says if I can prove P then I can prove Q. Now, suppose that we manage to give a proof of P. It follows from the $\Rightarrow$-elimination rule that Q is also provable.

Observe that in an introduction rule, the conclusion contains the logical connective associated with the rule, in this case, $\Rightarrow$; this justifies the terminology "introduction". On the other hand, in an elimination rule, the logical connective associated with the rule is gone (although it may still appear in Q). The other inference rules for $\wedge$, $\vee$, and the like, follow this pattern of introduction and elimination.

2.4 Examples of Proof Trees

(a) Here is a proof tree for $P \Rightarrow P$:

$$\dfrac{\dfrac{P^x}{P}}{P \Rightarrow P}\, x$$

So, $P \Rightarrow P$ is provable; this is the least we should expect from our proof system! Note that

$$\dfrac{P^x}{P \Rightarrow P}\, x$$

is also a valid proof tree for $P \Rightarrow P$, because the one-node tree labeled with P is a deduction tree.

(b) Here is a proof tree for $(P \Rightarrow Q) \Rightarrow ((Q \Rightarrow R) \Rightarrow (P \Rightarrow R))$:

$$\dfrac{\dfrac{\dfrac{\dfrac{(Q \Rightarrow R)^y \qquad \dfrac{(P \Rightarrow Q)^z \qquad P^x}{Q}}{R}}{P \Rightarrow R}\, x}{(Q \Rightarrow R) \Rightarrow (P \Rightarrow R)}\, y}{(P \Rightarrow Q) \Rightarrow ((Q \Rightarrow R) \Rightarrow (P \Rightarrow R))}\, z$$

In order to better appreciate the difference between a deduction tree and a proof tree, consider the following two examples.

1. The tree below is a deduction tree beause two of its leaves are labeled with the premises $P \Rightarrow Q$ and $Q \Rightarrow R$, that have not been discharged yet. So this tree represents a deduction of $P \Rightarrow R$ from the set of premises $\Gamma = \{P \Rightarrow Q, Q \Rightarrow R\}$ but it is *not a proof tree* because $\Gamma \neq \emptyset$. However, observe that the original premise P, labeled x, has been discharged.

$$\dfrac{\dfrac{Q \Rightarrow R \qquad \dfrac{P \Rightarrow Q \qquad P^x}{Q}}{R}}{P \Rightarrow R}\, x$$

2. The next tree was obtained from the previous one by applying the $\Rightarrow$-introduction rule which triggered the discharge of the premise $Q \Rightarrow R$ labeled y, which is no longer active. However, the premise $P \Rightarrow Q$ is still active (has not been discharged yet), so the tree below is a deduction tree of $(Q \Rightarrow R) \Rightarrow (P \Rightarrow R)$ from the set of premises $\Gamma = \{P \Rightarrow Q\}$. It is not yet a proof tree inasmuch as $\Gamma \neq \emptyset$.

$$\dfrac{\dfrac{\dfrac{(Q \Rightarrow R)^y \qquad \dfrac{P \Rightarrow Q \qquad P^x}{Q}}{R}}{P \Rightarrow R}\; x}{(Q \Rightarrow R) \Rightarrow (P \Rightarrow R)}\; y$$

Finally, one more application of the $\Rightarrow$-introduction rule discharged the premise $P \Rightarrow Q$, at last, yielding the proof tree in (b).

(c) This example illustrates the fact that different proof trees may arise from the same set of premises $\{P, Q\}$. For example, here are proof trees for $Q \Rightarrow (P \Rightarrow P)$ and $P \Rightarrow (Q \Rightarrow P)$:

$$\dfrac{\dfrac{\dfrac{P^x, Q^y}{P}}{P \Rightarrow P}\; x}{Q \Rightarrow (P \Rightarrow P)}\; y$$

and

$$\dfrac{\dfrac{\dfrac{P^x, Q^y}{P}}{Q \Rightarrow P}\; y}{P \Rightarrow (Q \Rightarrow P)}\; x$$

Similarly, there are six proof trees with a conclusion of the form

$$A \Rightarrow (B \Rightarrow (C \Rightarrow P))$$

begining with the deduction

$$\dfrac{P^x, Q^y, R^z}{P}$$

where A, B, C correspond to the six permutations of the premises P, Q, R.

Note that we would not have been able to construct the above proofs if Axiom (ii),

$$\dfrac{\Gamma, P}{P}\quad ,$$

were not available. We need a mechanism to "stuff" more premises into the leaves of our deduction trees in order to be able to discharge them later on. We may also view Axiom (ii) as a *weakening rule* whose purpose is to weaken a set of assumptions. Even though we are assuming all of the proposition in Γ and P, we only use the assumption P. The necessity of allowing multisets of premises is illustrated by the following proof of the proposition $P \Rightarrow (P \Rightarrow (Q \Rightarrow (Q \Rightarrow (P \Rightarrow P))))$:

$$\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{P^u, P^v, P^y, Q^w, Q^x}{P}}{P \Rightarrow P}\;y}{Q \Rightarrow (P \Rightarrow P)}\;x}{Q \Rightarrow (Q \Rightarrow (P \Rightarrow P))}\;w}{P \Rightarrow (Q \Rightarrow (Q \Rightarrow (P \Rightarrow P)))}\;v}{P \Rightarrow (P \Rightarrow (Q \Rightarrow (Q \Rightarrow (P \Rightarrow P))))}\;u$$

(d) In the next example which shows a proof of

$$\big(A \Rightarrow (B \Rightarrow C)\big) \Rightarrow \big((A \Rightarrow B) \Rightarrow (A \Rightarrow C)\big),$$

the two occurrences of A labeled x are *discharged simultaneously*:

$$\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{(A \Rightarrow (B \Rightarrow C))^z \qquad A^x}{B \Rightarrow C} \qquad \cfrac{(A \Rightarrow B)^y \qquad A^x}{B}}{C}}{A \Rightarrow C}\;x}{(A \Rightarrow B) \Rightarrow (A \Rightarrow C)}\;y}{\big(A \Rightarrow (B \Rightarrow C)\big) \Rightarrow \big((A \Rightarrow B) \Rightarrow (A \Rightarrow C)\big)}\;z$$

(e) In contrast to Example (d), in the proof tree below with conclusion

$$A \Rightarrow \Big(\big(A \Rightarrow (B \Rightarrow C)\big) \Rightarrow \big((A \Rightarrow B) \Rightarrow (A \Rightarrow C)\big)\Big),$$

the two occurrences of A are *discharged separately*. To this effect, they are labeled differently.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{(A \Rightarrow (B \Rightarrow C))^z \qquad A^x}{B \Rightarrow C} \qquad \dfrac{(A \Rightarrow B)^y \qquad A^t}{B}}{C}}{A \Rightarrow C}\; x}{(A \Rightarrow B) \Rightarrow (A \Rightarrow C)}\; y}{\big(A \Rightarrow (B \Rightarrow C)\big) \Rightarrow \big((A \Rightarrow B) \Rightarrow (A \Rightarrow C)\big)}\; z}{A \Rightarrow \Big(\big(A \Rightarrow (B \Rightarrow C)\big) \Rightarrow \big((A \Rightarrow B) \Rightarrow (A \Rightarrow C)\big)\Big)}\; t$$

How do we find these proof trees? Well, we could try to enumerate all possible proof trees systematically and see if a proof of the desired conclusion turns up. Obviously, this is a very inefficient procedure and moreover, how do we know that all possible proof trees will be generated and how do we know that such a method will terminate after a finite number of steps (what if the proposition proposed as a conclusion of a proof is not provable)?

Finding an algorithm to decide whether a proposition is provable is a very difficult problem and for sets of propositions with enough "expressive power" (such as propositions involving first-order quantifiers), it can be shown that there is **no** procedure that will give an answer in all cases and terminate in a finite number of steps for all possible input propositions. We come back to this point in Section 2.12. However, for the system $\mathcal{N}_m^{\Rightarrow}$, such a procedure exists but it is not easy to prove that it terminates in all cases and in fact, it can take a very long time.

What we did, and we strongly advise our readers to try it when they attempt to construct proof trees, is to construct the proof tree from the *bottom up*, starting from the proposition labeling the root, rather than top-down, that is, starting from the leaves. During this process, whenever we are trying to prove a proposition $P \Rightarrow Q$, we use the $\Rightarrow$-introduction rule backward, that is, we add P to the set of active premises and we try to prove Q from this new set of premises. At some point, we get stuck with an atomic proposition, say R. Call the resulting deduction $\mathcal{D}_{bu}$; note that R is the only active (undischarged) premise of $\mathcal{D}_{bu}$ and the node labeled R immediately below it plays a special role; we call it the special node of $\mathcal{D}_{bu}$.

Here is an illustration of this method for Example (d). At the end of the bottom-up process, we get the deduction tree $\mathcal{D}_{bu}$:

$$\dfrac{\dfrac{\dfrac{\dfrac{(A \Rightarrow (B \Rightarrow C))^z \qquad (A \Rightarrow B)^y \qquad A^x \qquad C}{C}}{A \Rightarrow C}\; x}{(A \Rightarrow B) \Rightarrow (A \Rightarrow C)}\; y}{\big(A \Rightarrow (B \Rightarrow C)\big) \Rightarrow \big((A \Rightarrow B) \Rightarrow (A \Rightarrow C)\big)}\; z$$

In the above deduction tree the proposition $R = C$ is the only active (undischarged) premise. To turn the above deduction tree into a proof tree we need to construct a deduction of C *from the premises other than* C. This is a more creative step which can be quite difficult. The trick is now to *switch strategies and start building a proof tree top-down*, starting from the leaves, using the $\Rightarrow$-elimination rule. If everything works out well, we get a deduction with root R, say $\mathcal{D}_{td}$, and then we glue this deduction $\mathcal{D}_{td}$ to the deduction $\mathcal{D}_{bu}$ in such a way that the root of $\mathcal{D}_{td}$ is identified with the special node of $\mathcal{D}_{bu}$ labeled R.

We also have to make sure that all the discharged premises are linked to the correct instance of the $\Rightarrow$-introduction rule that caused them to be discharged. One of the difficulties is that during the bottom-up process, we don't know how many copies of a premise need to be discharged in a single step. We only find out how many copies of a premise need to be discharged during the top-down process.

Going back to our example, at the end of the top-down process, we get the deduction tree $\mathcal{D}_{td}$:

$$\dfrac{\dfrac{A \Rightarrow (B \Rightarrow C) \qquad A}{B \Rightarrow C} \qquad \dfrac{A \Rightarrow B \qquad A}{B}}{C}$$

Finally, after gluing $\mathcal{D}_{td}$ on top of $\mathcal{D}_{bu}$ (which has the correct number of premises to be discharged), we get our proof tree:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{(A \Rightarrow (B \Rightarrow C))^z \qquad A^x}{B \Rightarrow C} \qquad \dfrac{(A \Rightarrow B)^y \qquad A^x}{B}}{C}}{A \Rightarrow C}\ x}{(A \Rightarrow B) \Rightarrow (A \Rightarrow C)}\ y}{\big(A \Rightarrow (B \Rightarrow C)\big) \Rightarrow \big((A \Rightarrow B) \Rightarrow (A \Rightarrow C)\big)}\ z$$

(f) The following example shows that proofs may be redundant. The proposition $P \Rightarrow ((P \Rightarrow Q) \Rightarrow Q)$ has the following proof.

$$\dfrac{\dfrac{\dfrac{(P \Rightarrow Q)^x \qquad P^y}{Q}}{(P \Rightarrow Q) \Rightarrow Q}\ x}{P \Rightarrow ((P \Rightarrow Q) \Rightarrow Q)}\ y$$

Now, say P is the proposition $R \Rightarrow R$, which has the proof

$$\dfrac{\dfrac{R^z}{R}}{R \Rightarrow R}\ z$$

Using $\Rightarrow$-elimination, we obtain a proof of $((R \Rightarrow R) \Rightarrow Q) \Rightarrow Q$ from the proof of $(R \Rightarrow R) \Rightarrow (((R \Rightarrow R) \Rightarrow Q) \Rightarrow Q)$ and the proof of $R \Rightarrow R$ shown above:

$$\dfrac{\dfrac{\dfrac{\dfrac{((R \Rightarrow R) \Rightarrow Q)^x \qquad (R \Rightarrow R)^y}{Q}}{((R \Rightarrow R) \Rightarrow Q) \Rightarrow Q}\,x}{(R \Rightarrow R) \Rightarrow (((R \Rightarrow R) \Rightarrow Q) \Rightarrow Q)}\,y \qquad \dfrac{\dfrac{R^z}{R}}{R \Rightarrow R}\,z}{((R \Rightarrow R) \Rightarrow Q) \Rightarrow Q}$$

Note that the above proof is *redundant*. The deduction tree shown in blue has the proposition $((R \Rightarrow R) \Rightarrow Q) \Rightarrow Q$ as conclusion but the proposition $R \Rightarrow R$ is introduced in the step labeled y and immediately eliminated in the next step. A more direct proof can be obtained as follows. Undo the last $\Rightarrow$-introduction (involving the the proposition $R \Rightarrow R$ and the tag y) in the proof of $(R \Rightarrow R) \Rightarrow (((R \Rightarrow R) \Rightarrow Q) \Rightarrow Q)$ obtaining the deduction tree shown in blue above

$$\dfrac{\dfrac{((R \Rightarrow R) \Rightarrow Q)^x \qquad R \Rightarrow R}{Q}}{((R \Rightarrow R) \Rightarrow Q) \Rightarrow Q}\,x$$

and then glue the proof of $R \Rightarrow R$ on top of the leaf $R \Rightarrow R$, obtaining the desired proof of $((R \Rightarrow R) \Rightarrow Q) \Rightarrow Q$:

$$\dfrac{\dfrac{((R \Rightarrow R) \Rightarrow Q)^x \qquad \dfrac{\dfrac{R^z}{R}}{R \Rightarrow R}\,z}{Q}}{((R \Rightarrow R) \Rightarrow Q) \Rightarrow Q}\,x$$

In general, one has to exercise care with the label variables. It may be necessary to rename some of these variables to avoid clashes. What we have above is an example of *proof substitution* also called *proof normalization*. We come back to this topic in Section 2.12.

While it is necessary to allow multisets of premises as shown in Example (c), our definition allows undesirable proof trees such as

$$\dfrac{\dfrac{\dfrac{P^x, P^x, Q^y, Q^y}{P}}{P \Rightarrow P}\,x}{Q \Rightarrow (P \Rightarrow P)}\,y$$

in which the two occurrences of P labeled x are discharged at the same time and the two occurrences of Q labeled y are discharged at the same time. Obviously, the above proof tree is equivalent to the proof tree

$$\cfrac{\cfrac{\cfrac{P^x, Q^y}{P}}{P \Rightarrow P}\, x}{Q \Rightarrow (P \Rightarrow P)}\, y$$

We leave it as an exercise to show that we can restrict ourselves to deduction trees and proof trees in which the labels of propositions appearing as premises of Rule Axioms (ii) are *all distinct*.

2.5 A Gentzen-Style System for Natural Deduction

The process of discharging premises when constructing a deduction is admittedly a bit confusing. Part of the problem is that a deduction tree really represents the last of a sequence of stages (corresponding to the application of inference rules) during which the current set of "active" premises, that is, those premises that have not yet been discharged (closed, cancelled) evolves (in fact, shrinks). Some mechanism is needed to keep track of which premises are no longer active and this is what this business of labeling premises with variables achieves. Historically, this is the first mechanism that was invented. However, Gentzen (in the 1930s) came up with an alternative solution that is mathematically easier to handle. Moreover, it turns out that this notation is also better suited to computer implementations, if one wishes to implement an automated theorem prover.

The point is to keep a record of all undischarged assumptions at every stage of the deduction. Thus, a deduction is now a tree whose nodes are labeled with pairs of the form $\langle \Gamma, P \rangle$, where P is a proposition, and Γ is a record of all undischarged assumptions at the stage of the deduction associated with this node.

Instead of using the notation $\langle \Gamma, P \rangle$, which is a bit cumbersome, Gentzen used expressions of the form $\Gamma \to P$, called *sequents*

It should be noted that the symbol $\to$ is used as a *separator* between the left-hand side Γ, called the *antecedent*, and the right-hand side P, called the *conclusion* (or *succedent*) and any other symbol could be used. Of course $\to$ is reminiscent of implication but we should not identify $\to$ and $\Rightarrow$. Still, it turns out that a sequent $\Gamma \to P$ is provable if and only if $(P_1 \wedge \cdots \wedge P_m) \Rightarrow P$ is provable, where $\Gamma = (P_1, \ldots, P_m)$.

During the construction of a deduction tree, it is necessary to discharge packets of assumptions consisting of one or more occurrences of the same proposition. To this effect, it is convenient to tag packets of assumptions with labels, in order to discharge the propositions in these packets in a single step. We use variables for the labels, and a packet labeled with x consisting of occurrences of the proposition P is written as $x\colon P$.

Definition 2.2. A *sequent* is an expression $\Gamma \to P$, where Γ is any finite *set* of the form $\{x_1 \colon P_1, \ldots, x_m \colon P_m\}$ called a *context*, where *the x_i are pairwise distinct* (but the P_i need not be distinct). Given $\Gamma = \{x_1 \colon P_1, \ldots, x_m \colon P_m\}$, the notation $\Gamma, x \colon P$ is only well defined when $x \neq x_i$ for all i, $1 \leq i \leq m$, in which case it denotes the set $\{x_1 \colon P_1, \ldots, x_m \colon P_m, x \colon P\}$. Given two contexts Γ and Δ, the context $\Gamma \cup \Delta$ is the union of the sets of pairs $(x_i \colon P_i)$ in Γ and the set of pairs $(y_k \colon Q_j)$ in Δ, provided that if $x \colon P \in \Gamma$ and $x \colon Q \in \Delta$ for the same variable x, then $P = Q$. In this case we say that Γ and Δ are *consistent*. So if $x \colon P$ occurs both in Γ and Δ, then $x \colon P$ also occurs in $\Gamma \cup \Delta$ (once).

One can think of a context $\Gamma = \{x_1 \colon P_1, \ldots, x_m \colon P_m\}$ as a set of type declarations for the variables $x_1, \ldots, x_m$ (x_i has type P_i). It should be noted that in the Prawitz-style formalism for proof trees, premises are treated as *multisets*, but in the Genten-style formalism, premises are *sets* of tagged pairs.

Using sequents, the axioms and rules of Definition 2.3 are now expressed as follows.

Definition 2.3. The axioms and inference rules of the system $\mathcal{NG}_m^{\Rightarrow}$ (*implicational logic, Gentzen-sequent style (the $\mathcal{G}$ in $\mathcal{NG}$ stands for Gentzen)*) are listed below:

$$\Gamma, x \colon P \to P \quad \text{(Axioms)}$$

$$\frac{\Gamma, x \colon P \to Q}{\Gamma \to P \Rightarrow Q} \quad (\Rightarrow\text{-}intro)$$

$$\frac{\Gamma \to P \Rightarrow Q \quad \Delta \to P}{\Gamma \cup \Delta \to Q} \quad (\Rightarrow\text{-}elim)$$

In an application of the rule ($\Rightarrow$-*intro*), observe that in the lower sequent, the proposition P (labeled x) is deleted from the list of premises occurring on the left-hand side of the arrow in the upper sequent. We say that the proposition P that appears as a hypothesis of the deduction is *discharged* (or *closed*). In the rule ($\Rightarrow$-*elim*), it is assumed that Γ and Δ are consistent contexts. A *deduction tree* is either a one-node tree labeled with an axiom or a tree constructed using the above inference rules. A *proof tree* is a deduction tree whose conclusion is a sequent with *an empty set of premises* (a sequent of the form $\to P$).

It is important to note that the ability to label packets consisting of occurrences of the same proposition with different labels is essential in order to be able to have control over which groups of packets of assumptions are discharged simultaneously. Equivalently, we could avoid tagging packets of assumptions with variables if we assume that in a sequent $\Gamma \to C$, the expression Γ is a *multiset* of propositions.

Let us display the proof tree for the second proof tree in Example (c) in our new Gentzen-sequent system. The orginal proof tree is

$$\cfrac{\cfrac{\cfrac{P^x, Q^y}{P}}{Q \Rightarrow P}\ {\scriptstyle y}}{P \Rightarrow (Q \Rightarrow P)}\ {\scriptstyle x}$$

and the corresponding proof tree in our new system is

$$\dfrac{\dfrac{x\colon P, y\colon Q \to P}{x\colon P \to Q \Rightarrow P}}{\to P \Rightarrow (Q \Rightarrow P)}$$

Below we show a proof of the first proposition of Example (d) given above in our new system.

$$\dfrac{\dfrac{\dfrac{\dfrac{z\colon A \Rightarrow (B \Rightarrow C) \to A \Rightarrow (B \Rightarrow C) \quad x\colon A \to A}{z\colon A \Rightarrow (B \Rightarrow C), x\colon A \to B \Rightarrow C} \qquad \dfrac{y\colon A \Rightarrow B \to A \Rightarrow B \quad x\colon A \to A}{y\colon A \Rightarrow B, x\colon A \to B}}{z\colon A \Rightarrow (B \Rightarrow C), y\colon A \Rightarrow B, x\colon A \to C}}{\dfrac{z\colon A \Rightarrow (B \Rightarrow C), y\colon A \Rightarrow B \to A \Rightarrow C}{z\colon A \Rightarrow (B \Rightarrow C) \to (A \Rightarrow B) \Rightarrow (A \Rightarrow C)}}}{\to \big(A \Rightarrow (B \Rightarrow C)\big) \Rightarrow \big((A \Rightarrow B) \Rightarrow (A \Rightarrow C)\big)}$$

It is not hard to design an algorithm that converts a deduction tree (or a proof tree) in the system $\mathcal{N}_m^{\Rightarrow}$ into a deduction tree (or a proof tree) in the system $\mathcal{NG}_m^{\Rightarrow}$, and vice-versa. In both cases the underlying tree is exactly the same and there is a bijection between the sets of undischarged premises in both representations.

After experimenting with the construction of proofs, one gets the feeling that every proof can be simplified to a "unique minimal" proof, if we define "minimal" in a suitable sense, namely, that a minimal proof never contains an elimination rule immediately following an introduction rule (for more on this, see Section 2.12). Then it turns out that to define the notion of uniqueness of proofs, the second version is preferable. However, it is important to realize that in general, a proposition may possess distinct minimal proofs.

In principle, it does not matter which of the two systems $\mathcal{N}_m^{\Rightarrow}$ or $\mathcal{NG}_m^{\Rightarrow}$ we use to construct deductions; it is basically a matter of taste. The Prawitz-style system $\mathcal{N}_m^{\Rightarrow}$ produces proofs that are closer to the informal proofs that humans construct. One the other hand, the Gentzen-style system $\mathcal{NG}_m^{\Rightarrow}$ is better suited for implementing theorem provers. My experience is that I make fewer mistakes with the Gentzen-sequent style system $\mathcal{NG}_m^{\Rightarrow}$.

We now describe the inference rules dealing with the connectives $\wedge$, $\vee$ and $\perp$.

2.6 Adding $\wedge$, $\vee$, $\perp$; The Proof Systems $\mathcal{N}_c^{\Rightarrow,\wedge,\vee,\perp}$ and $\mathcal{NG}_c^{\Rightarrow,\wedge,\vee,\perp}$

In this section we describe the proof rules for all the connectives of propositional logic both in Prawitz-style and in Gentzen-style. As we said earlier, the rules of the Prawitz-style system are closer to the rules that human use informally, and the rules of the Gentzen-style system are more convenient for computer implementations of theorem provers.

The rules involving $\perp$ are not as intuitively justifed as the other rules. In fact, in the early 1900s, some mathematicians especially L. Brouwer (1881–1966), questioned the validity of the proof-by-contradiction rule, among other principles. This led to the idea that it may be useful to consider proof systems of different strength. The weakest (and considered the safest) system is called *minimal logic*. This system rules out the $\perp$-elimination rule (the ability to deduce any proposition once a contradiction has been established) and the proof–by–contradiction rule. *Intuitionistic logic* rules out the proof–by–contradiction rule, and *classical logic* allows all the rules. Most people use classical logic, but intuitionistic logic is an interesting alternative because it is more constructive. We will elaborate on this point later. Minimal logic is just too weak.

Recall that $\neg P$ is an abbreviation for $P \Rightarrow \perp$.

Definition 2.4. The axioms, inference rules, and deduction trees for *(propositional) classical logic* are defined as follows. In the axioms and rules below, Γ, Δ, or Λ may be empty; P, Q, R denote arbitrary propositions built up from the atoms in **PS**; $\mathcal{D}$, $\mathcal{D}_1$, $\mathcal{D}_2$ denote deductions, possibly a one-node tree; and all the premises labeled x or y are discharged.

Axioms:

(i) Every one-node tree labeled with a single proposition P is a deduction tree for P with set of premises $\{P\}$.

(ii) The tree

$$\frac{\Gamma, P}{P}$$

is a deduction tree for P with multiset of premises Γ, P.

The **$\Rightarrow$-introduction rule**:

If $\mathcal{D}$ is a deduction of Q from the premises in Γ and one or more occurrences of the proposition P, then

$$\begin{array}{c} \Gamma, P^x \\ \mathcal{D} \\ Q \\ \hline P \Rightarrow Q \end{array}\ x$$

is a deduction tree for $P \Rightarrow Q$ from Γ. Note that this inference rule has the additional effect of discharging a nonempty set of occurrences of the premise P (which label leaves of the deduction $\mathcal{D}$). These occurrences are tagged with a new variable x, and the tag x is also placed immediately to the right of the inference bar. This proof rule corresponds to Proof Template 1.2 (Implication–Intro).

The **$\Rightarrow$-elimination rule (or** *modus ponens***)**:

If $\mathcal{D}_1$ is a deduction tree for $P \Rightarrow Q$ from the premises Γ, and $\mathcal{D}_2$ is a deduction for P from the premises Δ, then

$$\dfrac{\begin{matrix}\Gamma \\ \mathcal{D}_1 \\ P \Rightarrow Q\end{matrix} \qquad \begin{matrix}\Delta \\ \mathcal{D}_2 \\ P\end{matrix}}{Q}$$

is a deduction tree for Q from the premises in the multiset Γ, Δ. This proof rule corresponds to Proof Template 1.3 (Implication–Elim).

The $\wedge$**-introduction rule**:

If $\mathcal{D}_1$ is a deduction tree for P from the premises Γ, and $\mathcal{D}_2$ is a deduction for Q from the premises Δ, then

$$\dfrac{\begin{matrix}\Gamma \\ \mathcal{D}_1 \\ P\end{matrix} \qquad \begin{matrix}\Delta \\ \mathcal{D}_2 \\ Q\end{matrix}}{P \wedge Q}$$

is a deduction tree for $P \wedge Q$ from the premises in the multiset Γ, Δ. This proof rule corresponds to Proof Template 1.8 (And–Intro).

The $\wedge$**-elimination rule**:

If $\mathcal{D}$ is a deduction tree for $P \wedge Q$ from the premises Γ, then

$$\dfrac{\begin{matrix}\Gamma \\ \mathcal{D} \\ P \wedge Q\end{matrix}}{P} \qquad\qquad \dfrac{\begin{matrix}\Gamma \\ \mathcal{D} \\ P \wedge Q\end{matrix}}{Q}$$

are deduction trees for P and Q from the premises Γ. This proof rule corresponds to Proof Template 1.9 (And–elim).

The $\vee$**-introduction rule**:

If $\mathcal{D}$ is a deduction tree for P or for Q from the premises Γ, then

$$\dfrac{\begin{matrix}\Gamma \\ \mathcal{D} \\ P\end{matrix}}{P \vee Q} \qquad\qquad \dfrac{\begin{matrix}\Gamma \\ \mathcal{D} \\ Q\end{matrix}}{P \vee Q}$$

are deduction trees for $P \vee Q$ from the premises in Γ. This proof rule corresponds to Proof Template 1.10 (Or–Intro).

The $\vee$**-elimination rule**:

If $\mathcal{D}_1$ is a deduction tree for $P \vee Q$ from the premises Γ, $\mathcal{D}_2$ is a deduction for R from the premises in the multiset Δ and one or more occurrences of P, and $\mathcal{D}_3$ is a deduction for R from the premises in the multiset Λ and one or more occurrences of Q, then

$$\frac{\begin{array}{ccc} \Gamma & \Delta, P^x & \Lambda, Q^y \\ \mathcal{D}_1 & \mathcal{D}_2 & \mathcal{D}_3 \\ P \vee Q & R & R \end{array}}{R}\; {\scriptstyle x,y}$$

is a deduction tree for R from the premises in the multiset Γ, Δ, Λ. A nonempty set of premises P in $\mathcal{D}_2$ labeled x and a nonempty set of premises Q in $\mathcal{D}_3$ labeled y are discharged. This proof rule corresponds to Proof Template 1.11 (Or–Elim).

The **$\perp$-elimination rule**:

If $\mathcal{D}$ is a deduction tree for $\perp$ from the premises Γ, then

$$\frac{\begin{array}{c} \Gamma \\ \mathcal{D} \\ \perp \end{array}}{P}$$

is a deduction tree for P from the premises Γ, for *any* proposition P. This proof rule corresponds to Proof Template 1.6 (Perp–Elim).

The **proof–by–contradiction rule** (also known as **reductio ad absurdum rule**, for short *RAA*):

If $\mathcal{D}$ is a deduction tree for $\perp$ from the premises in the multiset Γ and one or more occurrences of $\neg P$, then

$$\frac{\begin{array}{c} \Gamma, \neg P^x \\ \mathcal{D} \\ \perp \end{array}}{P}\; {\scriptstyle x}$$

is a deduction tree for P from the premises Γ. A nonempty set of premises $\neg P$ labeled x are discharged. This proof rule corresponds to Proof Template 1.7 (Proof–By–Contradiction Principle).

Because $\neg P$ is an abbreviation for $P \Rightarrow \perp$, the $\neg$-introduction rule is a special case of the $\Rightarrow$-introduction rule (with $Q = \perp$). However, it is worth stating it explicitly.

The **$\neg$-introduction rule**:

If $\mathcal{D}$ is a deduction tree for $\perp$ from the premises in the multiset Γ and one or more occurrences of P, then

$$\frac{\begin{array}{c} \Gamma, P^x \\ \mathcal{D} \\ \perp \end{array}}{\neg P}\; {\scriptstyle x}$$

is a deduction tree for $\neg P$ from the premises Γ. A nonempty set of premises P labeled x are discharged. This proof rule corresponds to Proof Template 1.4 (Negation–Intro).

The above rule can be viewed as a proof–by–contradiction principle applied to negated propositions.

Similarly, the $\neg$-elimination rule is a special case of $\Rightarrow$-elimination applied to $\neg P\ (= P \Rightarrow \perp)$ and P.

The **$\neg$-elimination rule**:

If $\mathcal{D}_1$ is a deduction tree for $\neg P$ from the premises Γ, and $\mathcal{D}_2$ is a deduction for P from the premises Δ, then

$$\frac{\begin{array}{cc} \Gamma & \Delta \\ \mathcal{D}_1 & \mathcal{D}_2 \\ \neg P & P \end{array}}{\perp}$$

is a deduction tree for $\perp$ from the premises in the multiset Γ, Δ. This proof rule corresponds to Proof Template 1.5 (Negation–Elim).

A *deduction tree* is either a one-node tree labeled with a single proposition or a tree constructed using the above axioms and inference rules. A *proof tree* is a deduction tree such that *all its premises* are discharged. The above proof system is denoted $\mathcal{N}_c^{\Rightarrow,\wedge,\vee,\perp}$ (here, the subscript c stands for *classical*).

The system obtained by removing the proof–by–contradiction (RAA) rule is called *(propositional) intuitionistic logic* and is denoted $\mathcal{N}_i^{\Rightarrow,\wedge,\vee,\perp}$. The system obtained by deleting both the $\perp$-elimination rule and the proof–by–contradiction rule is called *(propositional) minimal logic* and is denoted $\mathcal{N}_m^{\Rightarrow,\wedge,\vee,\perp}$

The version of $\mathcal{N}_c^{\Rightarrow,\wedge,\vee,\perp}$ in terms of Gentzen sequents is the following.

Definition 2.5. The axioms and inference rules of the system $\mathcal{NG}_c^{\Rightarrow,\wedge,\vee,\perp}$ (of *propositional classical logic, Gentzen-sequent style*) are listed below.

$$\Gamma, x\colon P \to P \quad \text{(Axioms)}$$

$$\frac{\Gamma, x\colon P \to Q}{\Gamma \to P \Rightarrow Q} \quad (\Rightarrow\text{-}intro)$$

$$\frac{\Gamma \to P \Rightarrow Q \quad \Delta \to P}{\Gamma \cup \Delta \to Q} \quad (\Rightarrow\text{-}elim)$$

$$\frac{\Gamma \to P \quad \Delta \to Q}{\Gamma \cup \Delta \to P \wedge Q} \quad (\wedge\text{-}intro)$$

$$\frac{\Gamma \to P \wedge Q}{\Gamma \to P} \quad (\wedge\text{-}elim) \qquad \frac{\Gamma \to P \wedge Q}{\Gamma \to Q} \quad (\wedge\text{-}elim)$$

$$\frac{\Gamma \to P}{\Gamma \to P \vee Q} \quad (\vee\text{-}intro) \qquad \frac{\Gamma \to Q}{\Gamma \to P \vee Q} \quad (\vee\text{-}intro)$$

$$\frac{\Gamma \to P \vee Q \quad \Delta, x\colon P \to R \quad \Lambda, y\colon Q \to R}{\Gamma \cup \Delta \cup \Lambda \to R} \quad (\vee\textit{-elim})$$

$$\frac{\Gamma \to \perp}{\Gamma \to P} \quad (\perp\textit{-elim})$$

$$\frac{\Gamma, x\colon \neg P \to \perp}{\Gamma \to P} \quad (\textit{by-contra})$$

$$\frac{\Gamma, x\colon P \to \perp}{\Gamma \to \neg P} \quad (\neg\text{-introduction})$$

$$\frac{\Gamma \to \neg P \quad \Delta \to P}{\Gamma \cup \Delta \to \perp} \quad (\neg\text{-elimination})$$

A *deduction tree* is either a one-node tree labeled with an axiom or a tree constructed using the above inference rules. A *proof tree* is a deduction tree whose conclusion is a sequent *with an empty set of premises* (a sequent of the form $\emptyset \to P$).

The rule ($\perp$-*elim*) is trivial (does nothing) when $P = \perp$, therefore from now on we assume that $P \neq \perp$. *Propositional minimal logic*, denoted $\mathcal{NG}_m^{\Rightarrow,\wedge,\vee,\perp}$, is obtained by dropping the ($\perp$-*elim*) and (*by-contra*) rules. *Propositional intuitionistic logic*, denoted $\mathcal{NG}_i^{\Rightarrow,\wedge,\vee,\perp}$, is obtained by dropping the (*by-contra*) rule.

When we say that a proposition P is *provable from* Γ, we mean that we can construct a proof tree whose conclusion is P and whose set of premises is Γ, in one of the systems $\mathcal{N}_c^{\Rightarrow,\wedge,\vee,\perp}$ or $\mathcal{NG}_c^{\Rightarrow,\wedge,\vee,\perp}$. Therefore, when we use the word "provable" unqualified, we mean provable in *classical logic*. If P is provable from Γ in one of the intuitionistic systems $\mathcal{N}_i^{\Rightarrow,\wedge,\vee,\perp}$ or $\mathcal{NG}_i^{\Rightarrow,\wedge,\vee,\perp}$, then we say *intuitionistically provable* (and similarly, if P is provable from Γ in one of the systems $\mathcal{N}_m^{\Rightarrow,\wedge,\vee,\perp}$ or $\mathcal{NG}_m^{\Rightarrow,\wedge,\vee,\perp}$, then we say *provable in minimal logic*). When P is provable from Γ, most people write $\Gamma \vdash P$, or $\vdash \Gamma \to P$, sometimes with the name of the corresponding proof system tagged as a subscript on the sign $\vdash$ if necessary to avoid ambiguities. When Γ is empty, we just say P is provable (provable in intuitionistic logic, and so on) and write $\vdash P$.

We treat *logical equivalence* as a derived connective: that is, we view $P \equiv Q$ as an abbreviation for $(P \Rightarrow Q) \wedge (Q \Rightarrow P)$. In view of the inference rules for $\wedge$, we see that to prove a logical equivalence $P \equiv Q$, we just have to prove both implications $P \Rightarrow Q$ and $Q \Rightarrow P$.

Since the only difference between the proof systems $\mathcal{N}_m^{\Rightarrow,\wedge,\vee,\perp}$ and $\mathcal{NG}_m^{\Rightarrow,\wedge,\vee,\perp}$ is the way in which they perform the bookkeeping of premises, it is intuitively clear that they are equivalent. However, they produce different kinds of proof so to be rigorous we must check that the proof systems $\mathcal{N}_m^{\Rightarrow,\wedge,\vee,\perp}$ and $\mathcal{NG}_m^{\Rightarrow,\wedge,\vee,\perp}$ (as well as the systems $\mathcal{N}_c^{\Rightarrow,\wedge,\vee,\perp}$ and $\mathcal{NG}_c^{\Rightarrow,\wedge,\vee,\perp}$) are equivalent. This is not hard to show but is a bit tedious; see Problem 2.14.

In view of the $\neg$-elimination rule, we may be tempted to interpret the provability of a negation $\neg P$ as "P is not provable." Indeed, if $\neg P$ and P were both provable, then $\perp$ would be provable. So, P should not be provable if $\neg P$ is. However, if P is not provable, then $\neg P$ is **not** provable in general. There are plenty of propositions such that neither P nor $\neg P$ is provable (for instance, P, with P an atomic proposition). Thus, the fact that P is not provable is not equivalent to the provability of $\neg P$ and we should not interpret $\neg P$ as "P is not provable."

Let us now make some (much-needed) comments about the above inference rules. There is no need to repeat our comments regarding the $\Rightarrow$-rules.

The $\vee$-introduction rule says that if P (or Q) has been proved from Γ, then $P \vee Q$ is also provable from Γ. Again, this makes sense intuitively as $P \vee Q$ is "weaker" than P and Q.

The $\vee$-elimination rule formalizes the *proof–by–cases* method. It is a more subtle rule. The idea is that if we know that in the case where P is already assumed to be provable and similarly in the case where Q is already assumed to be provable that we can prove R (also using premises in Γ), then if $P \vee Q$ is also provable from Γ, as we have "covered both cases," it should be possible to prove R from Γ only (i.e., the premises P and Q are discarded). For example, if $\mathrm{remain1}(n)$ is the proposition that asserts n is a natural number of the form $4k + 1$ and $\mathrm{remain3}(n)$ is the proposition that asserts n is a natural number of the form $4k + 3$ (for some natural number k), then we can prove the implication

$$(\mathrm{remain1}(n) \vee \mathrm{remain3}(n)) \Rightarrow \mathrm{odd}(n),$$

where $\mathrm{odd}(n)$ asserts that n is odd, namely, that n is of the form $2h + 1$ for some h.

To prove the above implication we first assume the premise, $\mathrm{remain1}(n) \vee \mathrm{remain3}(n)$. Next we assume each of the alternatives in this proposition. When we assume $\mathrm{remain1}(n)$, we have $n = 4k + 1 = 2(2k) + 1$ for some k, so n is odd. When we assume $\mathrm{remain3}(n)$, we have $n = 4k + 3 = 2(2k + 1) + 1$, so again, n is odd. By $\vee$-elimination, we conclude that $\mathrm{odd}(n)$ follows from the premise $\mathrm{remain1}(n) \vee \mathrm{remain3}(n)$, and by $\Rightarrow$-introduction, we obtain a proof of our implication.

The $\perp$-elimination rule formalizes the principle that once a false statement has been established, *then anything should be provable*.

The $\neg$-introduction rule is a proof–by–contradiction principle applied to negated propositions. In order to prove $\neg P$, we assume P and we derive a contradiction ($\perp$). It is a more restrictive principle than the classical proof–by–contradiction rule (RAA). Indeed, if the proposition P to be proven is not a negation (P is not of the form $\neg Q$), then the $\neg$-introduction rule cannot be applied. On the other hand, the classical proof-by-contradiction rule can be applied but we have to assume $\neg P$ as a premise. For further comments on the difference between the $\neg$-introduction rule and the classical proof–by–contradiction rule, see Section 2.7.

The proof–by–contradiction rule formalizes the method of proof by contradiction. That is, in order to prove that P can be deduced from some premises Γ, one may assume the negation $\neg P$ of P (intuitively, assume that P is false) and then derive a contradiction from Γ and $\neg P$ (i.e., derive falsity). Then P actually follows from Γ *without using* $\neg P$ *as a premise*, that is, $\neg P$ is discharged. For example, let us prove by contradiction that if n^2 is odd, then n itself must be odd, where n is a natural number.

Figure 2.2: L. E. J. Brouwer, 1881–1966

According to the proof–by–contradiction rule, let us assume that n is not odd, which means that n is even. (Actually, in this step we are using a property of the natural numbers that is proven by induction but let's not worry about that right now. A proof is given in Section 2.16.) But to say that n is even means that $n = 2k$ for some k and then $n^2 = 4k^2 = 2(2k^2)$, so n^2 is even, contradicting the assumption that n^2 is odd. By the proof–by–contradiction rule, we conclude that n must be odd.

Remark: If the proposition to be proven, P, is of the form $\neg Q$, then if we use the proof-by-contradiction rule, we have to assume the premise $\neg\neg Q$ and then derive a contradiction. Because we are using classical logic, we often make implicit use of the fact that $\neg\neg Q$ is equivalent to Q (see Proposition 2.2) and instead of assuming $\neg\neg Q$ as a premise, we assume Q as a premise. But then, observe that we are really using $\neg$-introduction.

In summary, when trying to prove a proposition P by contradiction, proceed as follows.

(1) If P is a negated formula (P is of the form $\neg Q$), then use the $\neg$-introduction rule; that is, assume Q as a premise and derive a contradiction.

(2) If P is *not* a negated formula, then use the the proof-by-contradiction rule; that is, assume $\neg P$ as a premise and derive a contradiction.

Most people, I believe, will be comfortable with the rules of minimal logic and will agree that they constitute a "reasonable" formalization of the rules of reasoning involving $\Rightarrow$, $\wedge$, and $\vee$. Indeed, these rules seem to express the intuitive meaning of the connectives $\Rightarrow$, $\wedge$, and $\vee$. However, some may question the two rules $\perp$-elimination and proof-by-contradiction. Indeed, their meaning is not as clear and, certainly, the proof-by-contradiction rule introduces a form of indirect reasoning that is somewhat worrisome.

The problem has to do with the meaning of disjunction and negation and more generally, with the notion of *constructivity* in mathematics. In fact, in the early 1900s, some mathematicians, especially L. Brouwer (1881–1966), questioned the validity of the proof-by-contradiction rule, among other principles.

Two specific cases illustrate the problem, namely, the propositions

$$P \vee \neg P \quad \text{and} \quad \neg\neg P \Rightarrow P.$$

As we show shortly, the above propositions are both provable in classical logic; see Proposition 2.1 and Proposition 2.2.

Now Brouwer and some mathematicians belonging to his school of thought (the so-called "intuitionists" or "constructivists") advocate that in order to prove a disjunction $P \vee Q$ (from some premises Γ) one has to either *exhibit* a proof of P or a proof or Q (from Γ). However, it can be shown that this fails for $P \vee \neg P$. The fact that $P \vee \neg P$ is provable (in classical logic) **does not** imply (in general) that either P is provable or that $\neg P$ is provable. That $P \vee \neg P$ is provable is sometimes called the *principle (or law) of the excluded middle*. In intuitionistic logic, $P \vee \neg P$ is **not** provable (in general). Of course, if one gives up the proof-by-contradiction rule, then fewer propositions become provable. On the other hand, one may claim that the propositions that remain provable have more constructive proofs and thus feel on safer grounds.

A similar controversy arises with the proposition $\neg\neg P \Rightarrow P$ (*double-negation rule*) If we give up the proof-by-contradiction rule, then this formula is no longer provable (i.e., $\neg\neg P$ is no longer equivalent to P). Perhaps this relates to the fact that if one says "I don't have no money," then this does not mean that this person has money. (Similarly with "I can't get no satisfaction.") However, note that one can still prove $P \Rightarrow \neg\neg P$ in minimal logic (try doing it). Even stranger, $\neg\neg\neg P \Rightarrow \neg P$ is provable in intuitionistic (and minimal) logic, so $\neg\neg\neg P$ and $\neg P$ are equivalent intuitionistically.

Remark: Suppose we have a deduction

$$\begin{array}{c} \Gamma, \neg P \\ \mathcal{D} \\ \perp \end{array}$$

as in the proof-by-contradiction rule. Then by $\neg$-introduction, we get a deduction of $\neg\neg P$ from Γ:

$$\dfrac{\begin{array}{c} \Gamma, \neg P^x \\ \mathcal{D} \\ \perp \end{array}}{\neg\neg P}\ x$$

So, if we knew that $\neg\neg P$ was equivalent to P (actually, if we knew that $\neg\neg P \Rightarrow P$ is provable), then the proof-by-contradiction rule would be justified as a valid rule (it follows from modus ponens). We can view the proof-by-contradiction rule as a sort of act of faith that consists in saying that if we can derive an inconsistency (i.e., chaos) by assuming the falsity of a statement P, then P has to hold in the first place. It not so clear that such an act of faith is justified and the intuitionists refuse to take it.

Constructivity in mathematics is a fascinating subject but it is a topic that is really outside the scope in this book. What we hope is that our brief and very incomplete discussion of constructivity issues made the reader aware that the rules of logic are not cast in stone and that, in particular, there isn't **only one** logic.

We feel safe in saying that most mathematicians work with classical logic and only a few of them have reservations about using the proof-by-contradiction rule. Nevertheless, intuitionistic logic has its advantages, especially when it comes to proving the correctess of programs (a branch of computer science). We come back to this point several times in this book.

In the rest of this section we make further useful remarks about (classical) logic and give some explicit examples of proofs illustrating the inference rules of classical logic. We begin by proving that $P \vee \neg P$ is provable in classical logic.

Proposition 2.1. *The proposition $P \vee \neg P$ is provable in classical logic.*

Proof. We prove that $P \vee (P \Rightarrow \perp)$ is provable by using the proof-by-contradiction rule as shown below:

$$\dfrac{\dfrac{((P \vee (P \Rightarrow \perp)) \Rightarrow \perp)^y \qquad \dfrac{\dfrac{\dfrac{((P \vee (P \Rightarrow \perp)) \Rightarrow \perp)^y \qquad \dfrac{P^x}{P \vee (P \Rightarrow \perp)}\ \vee\text{-intro}}{\perp}}{P \Rightarrow \perp}\ {\scriptstyle x}\ (\neg\text{-intro})}{P \vee (P \Rightarrow \perp)}\ \vee\text{-intro}}{\perp}}{P \vee (P \Rightarrow \perp)}\ {\scriptstyle y}\ (\text{by-contra})$$

$\square$

Next, we consider the equivalence of P and $\neg\neg P$.

Proposition 2.2. *The proposition $P \Rightarrow \neg\neg P$ is provable in minimal logic. The proposition $\neg\neg P \Rightarrow P$ is provable in classical logic. Therefore, in classical logic, P is equivalent to $\neg\neg P$.*

Proof. We leave that $P \Rightarrow \neg\neg P$ is provable in minimal logic as an exercise. Below is a proof of $\neg\neg P \Rightarrow P$ using the proof-by-contradiction rule:

$$\dfrac{\dfrac{\dfrac{((P \Rightarrow \perp) \Rightarrow \perp)^y \qquad (P \Rightarrow \perp)^x}{\perp}}{P}\ {\scriptstyle x}\ (\text{by-contra})}{((P \Rightarrow \perp) \Rightarrow \perp) \Rightarrow P}\ {\scriptstyle y}$$

$\square$

The next proposition shows why $\perp$ can be viewed as the "ultimate" contradiction.

Proposition 2.3. *In intuitionistic logic, the propositions $\perp$ and $P \wedge \neg P$ are equivalent for all P. Thus, $\perp$ and $P \wedge \neg P$ are also equivalent in classical propositional logic*

Proof. We need to show that both $\perp \Rightarrow (P \wedge \neg P)$ and $(P \wedge \neg P) \Rightarrow \perp$ are provable in intuitionistic logic. The provability of $\perp \Rightarrow (P \wedge \neg P)$ is an immediate consequence or $\perp$-elimination, with $\Gamma = \emptyset$. For $(P \wedge \neg P) \Rightarrow \perp$, we have the following proof.

$$\dfrac{\dfrac{\dfrac{(P \wedge \neg P)^x}{\neg P} \qquad \dfrac{(P \wedge \neg P)^x}{P}}{\perp}}{(P \wedge \neg P) \Rightarrow \perp}\ x$$

□

So, in intuitionistic logic (and also in classical logic), $\perp$ is equivalent to $P \wedge \neg P$ for all P. This means that $\perp$ is the "ultimate" contradiction; it corresponds to total inconsistency. By the way, we could have the bad luck that the system $\mathcal{N}_c^{\Rightarrow,\wedge,\vee,\perp}$ (or $\mathcal{N}_i^{\Rightarrow,\wedge,\vee,\perp}$ or even $\mathcal{N}_m^{\Rightarrow,\wedge,\vee,\perp}$) is *inconsistent*, that is, that $\perp$ is provable. Fortunately, this is not the case, although this is hard to prove. (It is also the case that $P \vee \neg P$ and $\neg\neg P \Rightarrow P$ are **not** provable in intuitionistic logic, but this too is hard to prove.)

2.7 Clearing Up Differences Among ¬-Introduction, ⊥-Elimination, and RAA

The differences between the rules, ¬-introduction, ⊥-elimination, and the proof-by-contradiction rule (RAA) are often unclear to the uninitiated reader and this tends to cause confusion. In this section we try to clear up some common misconceptions about these rules.

Confusion 1. Why is RAA not a special case of ¬-introduction?

$$\begin{array}{c} \Gamma, P^x \\ \mathcal{D} \\ \dfrac{\perp}{\neg P} \end{array}\ x\,(\neg\text{-intro}) \qquad\qquad \begin{array}{c} \Gamma, \neg P^x \\ \mathcal{D} \\ \dfrac{\perp}{P} \end{array}\ x\,(\text{RAA})$$

The only apparent difference between ¬-introduction (on the left) and RAA (on the right) is that in RAA, the premise P is negated but the conclusion is not, whereas in ¬-introduction the premise P is not negated but the conclusion is.

The important difference is that the conclusion of RAA is **not** negated. If we had applied ¬-introduction instead of RAA on the right, we would have obtained

$$\begin{array}{c} \Gamma, \neg P^x \\ \mathcal{D} \\ \dfrac{\perp}{\neg\neg P} \end{array}\ x\,(\neg\text{-intro})$$

where the conclusion would have been $\neg\neg P$ as opposed to P. However, as we already said earlier, $\neg\neg P \Rightarrow P$ is **not** provable intuitionistically. Consequently, RAA **is not** a special case of $\neg$-introduction. On the other hand, one may view $\neg$-introduction as a "constructive" version of RAA applying to negated propositions (propositions of the form $\neg P$).

Confusion 2. Is there any difference between $\perp$-elimination and RAA?

$$\dfrac{\begin{array}{c}\Gamma\\ \mathcal{D}\\ \perp\end{array}}{P}\;(\perp\text{-elim}) \qquad\qquad \dfrac{\begin{array}{c}\Gamma, \neg P^x\\ \mathcal{D}\\ \perp\end{array}}{P}\;{\scriptstyle x}\,(\text{RAA})$$

The difference is that $\perp$-elimination does not discharge any of its premises. In fact, RAA is a stronger rule that implies $\perp$-elimination as we now demonstate.

RAA implies $\perp$-Elimination

Suppose we have a deduction

$$\begin{array}{c}\Gamma\\ \mathcal{D}\\ \perp\end{array}$$

Then, for any proposition P, we can add the premise $\neg P$ to every leaf of the above deduction tree and we get the deduction tree

$$\begin{array}{c}\Gamma, \neg P\\ \mathcal{D}'\\ \perp\end{array}$$

We can now apply RAA to get the following deduction tree of P from Γ (because $\neg P$ is discharged), and this is just the result of $\perp$-elimination:

$$\dfrac{\begin{array}{c}\Gamma, \neg P^x\\ \mathcal{D}'\\ \perp\end{array}}{P}\;{\scriptstyle x}\,(\text{RAA})$$

The above considerations also show that RAA is obtained from $\neg$-introduction by adding the new rule of $\neg\neg$-*elimination* (also called *double-negation elimination*):

$$\dfrac{\begin{array}{c}\Gamma\\ \mathcal{D}\\ \neg\neg P\end{array}}{P}\;(\neg\neg\text{-elimination})$$

Some authors prefer adding the $\neg\neg$-elimination rule to intuitionistic logic instead of RAA in order to obtain classical logic. As we just demonstrated, the two additions are equivalent: by adding either RAA or $\neg\neg$-elimination to intuitionistic logic, we get classical logic.

There is another way to obtain RAA from the rules of intuitionistic logic, this time, using the propositions of the form $P \vee \neg P$. We saw in Proposition 2.1 that all formulae of the form $P \vee \neg P$ are provable in classical logic (using RAA).

Confusion 3. Are propositions of the form $P \vee \neg P$ provable in intuitionistic logic? The answer is **no**, which may be disturbing to some readers. In fact, it is quite difficult to prove that propositions of the form $P \vee \neg P$ are not provable in intuitionistic logic. One method consists in using the fact that intuitionistic proofs can be normalized (see Section 2.12 for more on normalization of proofs). Another method uses Kripke models (see Section 2.11 and van Dalen [51]).

Part of the difficulty in understanding at some intuitive level why propositions of the form $P \vee \neg P$ are not provable in intuitionistic logic is that the notion of truth based on the truth values **true** and **false** is deeply rooted in all of us. In this frame of mind, it seems ridiculous to question the provability of $P \vee \neg P$, because its truth value is **true** whether P is assigned the value **true** or **false**. Classical two-valued truth value semantics is too crude for intuitionistic logic.

Another difficulty is that it is tempting to equate the notion of truth and the notion of provability. Unfortunately, because classical truth values semantics is too crude for intuitionistic logic, there are propositions that are universally true (i.e., they evaluate to **true** for all possible truth assignments of the atomic letters in them) and yet they are **not** provable intuitionistically. The propositions $P \vee \neg P$ and $\neg\neg P \Rightarrow P$ are such examples.

One of the major motivations for advocating intuitionistic logic is that it yields proofs that are more constructive than classical proofs. For example, in classical logic, when we prove a disjunction $P \vee Q$, we generally can't conclude that either P or Q is provable, as exemplified by $P \vee \neg P$. A more interesting example involving a nonconstructive proof of a disjunction is given in Section 2.8. But in intuitionistic logic, from a proof of $P \vee Q$, it is possible to extract either a proof of P or a proof of Q (and similarly for existential statements; see Section 2.15). This property is not easy to prove. It is a consequence of the normal form for intuitionistic proofs (see Section 2.12).

In brief, besides being a fun intellectual game, intuitionistic logic is only an interesting alternative to classical logic if we care about the constructive nature of our proofs. But then we are forced to abandon the classical two-valued truth values semantics and adopt other semantics such as Kripke semantics. If we do not care about the constructive nature of our proofs and if we want to stick to two-valued truth values semantics, then we should stick to classical logic. Most people do that, so don't feel bad if you are not comfortable with intuitionistic logic.

One way to gauge how intuitionisic logic differs from classical logic is to ask what kind of propositions need to be added to intuitionisic logic in order to get classical logic. It turns out that if all the propositions of the form $P \vee \neg P$ are considered to be axioms, then RAA follows from some of the rules of intuitionistic logic.

RAA Holds in Intuitionistic Logic + All Axioms $P \vee \neg P$.
The proof involves a subtle use of the $\perp$-elimination and $\vee$-elimination rules which may be a bit puzzling. Assume, as we do when we use the proof-by-contradiction rule (RAA) that

we have a deduction

$$\begin{array}{c}\Gamma, \neg P\\ \mathcal{D}\\ \perp\end{array}$$

Here is the deduction tree demonstrating that RAA is a derived rule:

$$\dfrac{P \vee \neg P \qquad \dfrac{P^x}{P} \qquad \dfrac{\begin{array}{c}\Gamma, \neg P^y\\ \mathcal{D}\\ \perp\end{array}}{P}\ (\perp\text{-elim})}{P}\ {}_{x,y}\ (\vee\text{-elim})$$

At first glance, the rightmost subtree

$$\dfrac{\begin{array}{c}\Gamma, \neg P^y\\ \mathcal{D}\\ \perp\end{array}}{P}\ (\perp\text{-elim})$$

appears to use RAA and our argument looks circular. But this is not so because the premise $\neg P$ labeled y is *not* discharged in the step that yields P as conclusion; the step that yields P is a $\perp$-elimination step. The premise $\neg P$ labeled y is actually discharged by the $\vee$-elimination rule (and so is the premise P labeled x). So our argument establishing RAA is not circular after all.

In conclusion, intuitionistic logic is obtained from classical logic by *taking away the proof-by-contradiction rule (RAA)*. In this more restrictive proof system, we obtain more constructive proofs. In that sense, the situation is better than in classical logic. The major drawback is that we can't think in terms of classical truth values semantics anymore.

Conversely, classical logic is obtained from intuitionistic logic in at least three ways:

1. Add the proof-by-contradiction rule (RAA).

2. Add the $\neg\neg$-elimination rule.

3. Add all propositions of the form $P \vee \neg P$ as axioms.

2.8 De Morgan Laws and Other Rules of Classical Logic

In Section 1.7 we discussed the de Morgan laws. Now that we also know about intuitionistic logic we revisit these laws.

Proposition 2.4. *The following equivalences (de Morgan laws) are provable in classical logic.*

$$\neg(P \land Q) \equiv \neg P \lor \neg Q$$
$$\neg(P \lor Q) \equiv \neg P \land \neg Q.$$

In fact, $\neg(P \lor Q) \equiv \neg P \land \neg Q$ *and* $(\neg P \lor \neg Q) \Rightarrow \neg(P \land Q)$ *are provable in intuitionistic logic. The proposition* $(P \land \neg Q) \Rightarrow \neg(P \Rightarrow Q)$ *is provable in intuitionistic logic and* $\neg(P \Rightarrow Q) \Rightarrow (P \land \neg Q)$ *is provable in classical logic. Therefore,* $\neg(P \Rightarrow Q)$ *and* $P \land \neg Q$ *are equivalent in classical logic. Furthermore,* $P \Rightarrow Q$ *and* $\neg P \lor Q$ *are equivalent in classical logic and* $(\neg P \lor Q) \Rightarrow (P \Rightarrow Q)$ *is provable in intuitionistic logic.*

Proof. We only prove the very last part of Proposition 2.4 leaving the other parts as a series of exercises. Here is an intuitionistic proof of $(\neg P \lor Q) \Rightarrow (P \Rightarrow Q)$:

$$\dfrac{\dfrac{(\neg P \lor Q)^w \qquad \dfrac{\dfrac{\dfrac{\neg P^z \qquad P^x}{\perp}}{Q}}{P \Rightarrow Q}\,x \qquad \dfrac{\dfrac{P^y \qquad Q^t}{Q}}{P \Rightarrow Q}\,y}{P \Rightarrow Q}\,z,t}{(\neg P \lor Q) \Rightarrow (P \Rightarrow Q)}\,w$$

Here is a classical proof of $(P \Rightarrow Q) \Rightarrow (\neg P \lor Q)$:

$$\dfrac{\dfrac{\dfrac{(\neg(\neg P \lor Q))^y \qquad \dfrac{\dfrac{(P \Rightarrow Q)^z \qquad \dfrac{\dfrac{(\neg(\neg P \lor Q))^y \qquad \dfrac{\neg P^x}{\neg P \lor Q}}{\perp}}{P}\,x\ \mathrm{RAA}}{Q}}{\neg P \lor Q}}{\perp}}{\neg P \lor Q}\,y\ \mathrm{RAA}}{(P \Rightarrow Q) \Rightarrow (\neg P \lor Q)}\,z$$

The other proofs are left as exercises. ☐

Propositions 2.2 and 2.4 show a property that is very specific to classical logic, namely, that the logical connectives $\Rightarrow, \land, \lor, \neg$ are not independent. For example, we have $P \land Q \equiv \neg(\neg P \lor \neg Q)$, which shows that $\land$ can be expressed in terms of $\lor$ and $\neg$. In intuitionistic logic, $\land$ and $\lor$ cannot be expressed in terms of each other via negation.

The fact that the logical connectives $\Rightarrow, \wedge, \vee, \neg$ are not independent in classical logic suggests the following question. Are there propositions, written in terms of $\Rightarrow$ only, that are provable classically but not provable intuitionistically?

The answer is yes. For instance, the proposition $((P \Rightarrow Q) \Rightarrow P) \Rightarrow P$ (known as *Peirce's law*) is provable classically (do it) but it can be shown that it is not provable intuitionistically.

In addition to the proof-by-cases method and the proof-by-contradiction method, we also have the proof-by-contrapositive method valid in classical logic:

Proof-by-contrapositive rule:

$$\cfrac{\begin{array}{c}\Gamma, \neg Q^x \\ \mathcal{D} \\ \neg P\end{array}}{P \Rightarrow Q}\ {\scriptstyle x}$$

This rule says that in order to prove an implication $P \Rightarrow Q$ (from Γ), one may assume $\neg Q$ as proven, and then deduce that $\neg P$ is provable from Γ and $\neg Q$. This inference rule is valid in classical logic because we can construct the following deduction.

$$\cfrac{\cfrac{\cfrac{\begin{array}{c}\Gamma, \neg Q^x \\ \mathcal{D} \\ \neg P\end{array} \qquad P^y}{\perp}}{Q}\ {\scriptstyle x}\ \text{(by-contra)}}{P \Rightarrow Q}\ {\scriptstyle y}$$

As as example of the proof-by-contrapositive method, we prove that if an integer n^2 is even, then n must be even.

Observe that if an integer is not even, then it is odd (and vice versa). This fact may seem quite obvious but to prove it actually requires using *induction* (which we haven't officially met yet). A rigorous proof is given in Section 2.16.

Now the contrapositive of our statement is: if n is odd, then n^2 is odd. But to say that n is odd is to say that $n = 2k+1$ and then, $n^2 = (2k+1)^2 = 4k^2+4k+1 = 2(2k^2+2k)+1$, which shows that n^2 is odd.

As it is, because the above proof uses the proof-by-contrapositive method, it is not constructive. Thus, the question arises, is there a constructive proof of the above fact?

Indeed there is a constructive proof if we observe that every integer n is either even or odd but not both. Now, one might object that we just relied on the law of the excluded middle but there is a way to circumvent this problem by using *induction*; see Section 2.16 for a rigorous proof.

Now, because *an integer is odd iff it is not even*, we may proceed to prove that *if n^2 is even, then n is not odd*, by using our constructive version of the proof-by-contradiction principle, namely, $\neg$-introduction.

Therefore, assume that n^2 is even and that n is odd. Then $n = 2k + 1$, which implies that $n^2 = 4k^2 + 4k + 1 = 2(2k^2 + 2k) + 1$, an odd number, contradicting the fact that n^2 is assumed to be even.

The next proposition collects a list of equivalences involving conjunction and disjunction that are used all the time. Proofs of these propositions are left as exercises (see the problems).

Proposition 2.5. *All the propositions below are provable intuitionistically:*

$$\begin{aligned} P \vee P &\equiv P \\ P \wedge P &\equiv P \\ P \vee Q &\equiv Q \vee P \\ P \wedge Q &\equiv Q \wedge P. \end{aligned}$$

The last two assert the commutativity of $\vee$ and $\wedge$. We have distributivity of $\wedge$ over $\vee$ and of $\vee$ over $\wedge$:

$$\begin{aligned} P \wedge (Q \vee R) &\equiv (P \wedge Q) \vee (P \wedge R) \\ P \vee (Q \wedge R) &\equiv (P \vee Q) \wedge (P \vee R). \end{aligned}$$

We have associativity of $\wedge$ and $\vee$:

$$\begin{aligned} P \wedge (Q \wedge R) &\equiv (P \wedge Q) \wedge R \\ P \vee (Q \vee R) &\equiv (P \vee Q) \vee R. \end{aligned}$$

2.9 Formal Versus Informal Proofs

As we said before, *it is practically impossible to write formal proofs* (i.e., proofs written as proof trees using the rules of one of the systems presented earlier) of "real" statements that are not "toy propositions." This is because it would be extremely tedious and time-consuming to write such proofs and these proofs would be huge and thus very hard to read.

What we do instead is to construct "informal" proofs in which we still make use of the logical rules that we have presented but we take shortcuts and sometimes we even omit proof steps (some elimination rules, such as $\wedge$-elimination and some introduction rules, such as $\vee$-introduction) and we use a natural language (here, presumably, English) rather than formal symbols (we say "and" for $\wedge$, "or" for $\vee$, *etc.*). We refer the readetr to Section 1.8 for a discussion of these issues. We also urge our readers to read Chapter 3 of Gowers [26] which contains very illuminating remarks about the notion of proof in mathematics.

Here is a concrete example illustrating the usefulnes of auxiliary lemmas in constructing informal proofs.

Say we wish to prove the implication

$$\neg(P \wedge Q) \Rightarrow \big((\neg P \wedge \neg Q) \vee (\neg P \wedge Q) \vee (P \wedge \neg Q)\big). \qquad (*)$$

It can be shown that the above proposition is not provable intuitionistically, so we have to use the proof-by-contradiction method in our proof. One quickly realizes that any proof ends up re-proving basic properties of $\wedge$ and $\vee$, such as associativity, commutativity, idempotence, distributivity, and so on, some of the de Morgan laws, and that the complete proof is very large. However, if we allow ourselves to use the de Morgan laws as well as various basic properties of $\wedge$ and $\vee$, such as distributivity,

$$(A \wedge B) \vee C \equiv (A \wedge C) \vee (B \wedge C),$$

commutativity of $\wedge$ and $\vee$ ($A \wedge B \equiv B \wedge A$, $A \vee B \equiv B \vee A$), associativity of $\wedge$ and $\vee$ ($A \wedge (B \wedge C) \equiv (A \wedge B) \wedge C$, $A \vee (B \vee C) \equiv (A \vee B) \vee C$), and the idempotence of $\wedge$ and $\vee$ ($A \wedge A \equiv A$, $A \vee A \equiv A$), then we get

$$\begin{aligned}
(\neg P \wedge \neg Q) \vee (\neg P \wedge Q) \vee (P \wedge \neg Q) &\equiv (\neg P \wedge \neg Q) \vee (\neg P \wedge \neg Q) \\
&\qquad\qquad \vee (\neg P \wedge Q) \vee (P \wedge \neg Q) \\
&\equiv (\neg P \wedge \neg Q) \vee (\neg P \wedge Q) \\
&\qquad\qquad \vee (\neg P \wedge \neg Q) \vee (P \wedge \neg Q) \\
&\equiv (\neg P \wedge (\neg Q \vee Q)) \vee (\neg P \wedge \neg Q) \vee (P \wedge \neg Q) \\
&\equiv \neg P \vee (\neg P \wedge \neg Q) \vee (P \wedge \neg Q) \\
&\equiv \neg P \vee ((\neg P \vee P) \wedge \neg Q) \\
&\equiv \neg P \vee \neg Q,
\end{aligned}$$

where we make implicit uses of commutativity and associativity, and the fact that $R \wedge (P \vee \neg P) \equiv R$, and by de Morgan,

$$\neg(P \wedge Q) \equiv \neg P \vee \neg Q,$$

using auxiliary lemmas, we end up proving $(*)$ without too much pain.

2.10 Truth Value Semantics for Classical Logic Soundness and Completeness

In Section 1.9 we introduced the truth value semantics for classical propositional logic. The logical connectives $\Rightarrow$, $\wedge$, $\vee$, $\neg$ and $\equiv$ can be interpreted as Boolean functions, that is, functions whose arguments and whose values range over the set of *truth values*,

$$\mathbf{BOOL} = \{\mathbf{true}, \mathbf{false}\}.$$

These functions are given by the following *truth tables*.

P	Q	$P \Rightarrow Q$	$P \wedge Q$	$P \vee Q$	$\neg P$	$P \equiv Q$
true	**true**	**true**	**true**	**true**	**false**	**true**
true	**false**	**false**	**false**	**true**	**false**	**false**
false	**true**	**true**	**false**	**true**	**true**	**false**
false	**false**	**true**	**false**	**false**	**true**	**true**

Now, any proposition P built up over the set of atomic propositions **PS** (our propositional symbols) contains a finite set of propositional letters, say

$$\{P_1, \ldots, P_m\}.$$

If we assign some truth value (from **BOOL**) to each symbol P_i then we can "compute" the *truth value* of P under this assignment by using recursively using the truth tables above.

For example, the proposition $\mathbf{P}_1 \Rightarrow (\mathbf{P}_1 \Rightarrow \mathbf{P}_2)$, under the truth assignment v given by

$$\mathbf{P}_1 = \mathbf{true},\ \mathbf{P}_2 = \mathbf{false},$$

evaluates to **false**; see Section 1.9.

The values of a proposition can be determined by creating a *truth table*, in which a proposition is evaluated by computing recursively the truth values of its subexpressions. See Section 1.9.

The truth table of a proposition containing m variables has 2^m rows. When m is large, 2^m is very large, and computing the truth table of a proposition P may not be practically feasible. Even the problem of finding whether there is a truth assignment that makes P true is hard.

Definition 2.6. We say that a proposition P is *satisfiable* iff it evaluates to **true** for *some* truth assignment (taking values in **BOOL**) of the propositional symbols occurring in P and otherwise we say that it is *unsatisfiable*. A proposition P is *valid* (or a *tautology*) iff it evaluates to **true** for *all* truth assignments of the propositional symbols occurring in P.

Observe that a proposition P is valid if in the truth table for P *all* the entries in the column corresponding to P have the value **true**. The proposition P is satisfiable if some entry in the column corresponding to P has the value **true**.

The problem of deciding whether a proposition is satisfiable is called the *satisfiability problem* and is sometimes denoted by SAT. The problem of deciding whether a proposition is valid is called the *validity problem*.

For example, the proposition

$$P = (\mathbf{P}_1 \lor \neg\mathbf{P}_2 \lor \neg\mathbf{P}_3) \land (\neg\mathbf{P}_1 \lor \neg\mathbf{P}_3) \land (\mathbf{P}_1 \lor \mathbf{P}_2 \lor \mathbf{P}_4) \land (\neg\mathbf{P}_3 \lor \mathbf{P}_4) \land (\neg\mathbf{P}_1 \lor \mathbf{P}_4)$$

is satisfiable because it evaluates to **true** under the truth assignment $\mathbf{P}_1 = \mathbf{true}$, $\mathbf{P}_2 = \mathbf{false}$, $\mathbf{P}_3 = \mathbf{false}$, and $\mathbf{P}_4 = \mathbf{true}$. On the other hand, the proposition

$$Q = (\mathbf{P}_1 \lor \mathbf{P}_2 \lor \mathbf{P}_3) \land (\neg\mathbf{P}_1 \lor \mathbf{P}_2) \land (\neg\mathbf{P}_2 \lor \mathbf{P}_3) \land (\mathbf{P}_1 \lor \neg\mathbf{P}_3) \land (\neg\mathbf{P}_1 \lor \neg\mathbf{P}_2 \lor \neg\mathbf{P}_3)$$

is unsatisfiable as one can verify by trying all eight truth assignments for $\mathbf{P}_1$, $\mathbf{P}_2$, $\mathbf{P}_3$. The reader should also verify that the proposition

$$R = (\neg\mathbf{P}_1 \land \neg\mathbf{P}_2 \land \neg\mathbf{P}_3) \lor (\mathbf{P}_1 \land \neg\mathbf{P}_2) \lor (\mathbf{P}_2 \land \neg\mathbf{P}_3) \lor (\neg\mathbf{P}_1 \land \mathbf{P}_3) \lor (\mathbf{P}_1 \land \mathbf{P}_2 \land \mathbf{P}_3)$$

is valid (observe that the proposition R is the negation of the proposition Q).

The satisfiability problem is a famous problem in computer science because of its complexity. Try it; solving it is not as easy as you think. The difficulty is that if a proposition P contains n distinct propositional letters, then there are 2^n possible truth assignments and checking all of them is practically impossible when n is large.

In fact, the satisfiability problem turns out to be an *NP-complete* problem, a very important concept that you will learn about in a course on the theory of computation and complexity. Very good expositions of this kind of material are found in Hopcroft, Motwani, and Ullman [29] and Lewis and Papadimitriou [35]. The validity problem is also important and it is related to SAT. Indeed, it is easy to see that a proposition P is valid iff $\neg P$ is unsatisfiable.

What's the relationship between validity and provability in the system $\mathcal{N}_c^{\Rightarrow,\wedge,\vee,\perp}$ (or $\mathcal{NG}_c^{\Rightarrow,\wedge,\vee,\perp}$)?

Remarkably, in classical logic, *validity and provability are equivalent*.

In order to prove the above claim, we need to do two things:

(1) Prove that if a proposition P is provable in the system $\mathcal{N}_c^{\Rightarrow,\wedge,\vee,\perp}$ (or the system $\mathcal{NG}_c^{\Rightarrow,\wedge,\vee,\perp}$), then it is valid. This is known as *soundness* or *consistency* (of the proof system).

(2) Prove that if a proposition P is valid, then it has a proof in the system $\mathcal{N}_c^{\Rightarrow,\wedge,\vee,\perp}$ (or $\mathcal{NG}_c^{\Rightarrow,\wedge,\vee,\perp}$). This is known as the *completeness* (of the proof system).

In general, it is relatively easy to prove (1) but proving (2) can be quite complicated. In fact, some proof systems are *not* complete with respect to certain semantics. For instance, the proof system for intuitionistic logic $\mathcal{N}_i^{\Rightarrow,\wedge,\vee,\perp}$ (or $\mathcal{NG}_i^{\Rightarrow,\wedge,\vee,\perp}$) is *not complete* with respect to truth value semantics. As an example, $((P \Rightarrow Q) \Rightarrow P) \Rightarrow P$ (known as *Peirce's law*), is valid but it can be shown that it cannot be proven in intuitionistic logic.

In this book we content ourselves with soundness.

Proposition 2.6. *(Soundness of $\mathcal{N}_c^{\Rightarrow,\wedge,\vee,\perp}$ and $\mathcal{NG}_c^{\Rightarrow,\wedge,\vee,\perp}$) If a proposition P is provable in the system $\mathcal{N}_c^{\Rightarrow,\wedge,\vee,\perp}$ (or $\mathcal{NG}_c^{\Rightarrow,\wedge,\vee,\perp}$), then it is valid (according to the truth value semantics).*

Sketch of Proof. It is enough to prove that if there is a deduction of a proposition P from a set of premises Γ then for every truth assignment for which all the propositions in Γ evaluate to **true**, then P evaluates to **true**. However, this is clear for the axioms and every inference rule preserves that property.

Now if P is provable, a proof of P has an empty set of premises and so P evaluates to **true** for all truth assignments, which means that P is valid. $\square$

Theorem 2.7. *(Completeness of $\mathcal{N}_c^{\Rightarrow,\wedge,\vee,\perp}$ and $\mathcal{NG}_c^{\Rightarrow,\wedge,\vee,\perp}$) If a proposition P is valid (according to the truth value semantics), then P is provable in the system $\mathcal{N}_c^{\Rightarrow,\wedge,\vee,\perp}$ (or $\mathcal{NG}_c^{\Rightarrow,\wedge,\vee,\perp}$).*

Proofs of completeness for classical logic can be found in van Dalen [51] or Gallier [19] (but for a different proof system).

Soundness (Proposition 2.6) has a very useful consequence: in order to prove that a proposition P is *not provable*, it is enough to find a truth assignment for which P evaluates to **false**. We say that such a truth assignment is a *counterexample* for P (or that P can be *falsified*). For example, no propositional symbol $\mathbf{P}_i$ is provable because it is falsified by the truth assignment $\mathbf{P}_i = \mathbf{false}$.

The soundness of the proof system $\mathcal{N}_c^{\Rightarrow,\wedge,\vee,\perp}$ (or $\mathcal{NG}_c^{\Rightarrow,\wedge,\vee,\perp}$) also has the extremely important consequence that $\perp$ *cannot be proven* in this system, which means that *contradictory statements* cannot be derived.

This is by no means obvious at first sight, but reassuring. It is also possible to prove that the proof system $\mathcal{N}_c^{\Rightarrow,\wedge,\vee,\perp}$ is consistent (*i.e.*, $\perp$ cannot be proven) by purely proof-theoretic means involving proof normalization (See Section 2.12), but this requires a lot more work.

Note that completeness amounts to the fact that *every unprovable formula has a counterexample*. Also, in order to show that a proposition is classically provable, it suffices to compute its truth table and check that the proposition is valid. This may still be a lot of work, but it is a more "mechanical" process than attempting to find a proof.

For example, here is a truth table showing that $(\mathbf{P}_1 \Rightarrow \mathbf{P}_2) \equiv (\neg\mathbf{P}_1 \vee \mathbf{P}_2)$ is valid.

$\mathbf{P}_1$	$\mathbf{P}_2$	$\mathbf{P}_1 \Rightarrow \mathbf{P}_2$	$\neg\mathbf{P}_1 \vee \mathbf{P}_2$	$(\mathbf{P}_1 \Rightarrow \mathbf{P}_2) \equiv (\neg\mathbf{P}_1 \vee \mathbf{P}_2)$
true	**true**	**true**	**true**	**true**
true	**false**	**false**	**false**	**true**
false	**true**	**true**	**true**	**true**
false	**false**	**true**	**true**	**true**

Remark: Truth value semantics is not the right kind of semantics for intuitionistic logic; it is too coarse. A more subtle kind of semantics is required. Among the various semantics for intuitionistic logic, one of the most natural is the notion of the *Kripke model*. Then again, soundness and completeness hold for intuitionistic proof systems (see Section 2.11 and van Dalen [51]).

2.11 Kripke Models for Intuitionistic Logic Soundness and Completeness

In this section, we briefly describe the semantics of intuitionistic propositional logic in terms of Kripke models.

This section has been included to quench the thirst of those readers who can't wait to see what kind of decent semantics can be given for intuitionistic propositional logic and it can be safely omitted.

In classical truth value semantics based on $\mathbf{BOOL} = \{\mathbf{true}, \mathbf{false}\}$, we might say that truth is absolute. The idea of Kripke semantics is that there is a set of worlds (or states)

W together with a partial ordering $\leq$ on W, and that truth depends on in which world we are. Furthermore, as we "go up" from a world u to a world v with $u \leq v$, truth "can only increase," that is, whatever is true in world u remains true in world v. Also, the truth of some propositions, such as $P \Rightarrow Q$ or $\neg P$, depends on "future worlds." With this type of semantics, which is no longer absolute, we can capture exactly the essence of intuitionistic logic. We now make these ideas precise.

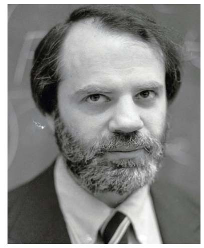


Figure 2.3: Saul Kripke, 1940–

Definition 2.7. A *Kripke model* for intuitionistic propositional logic is a pair $\mathcal{K} = (W, \varphi)$ where W is a partially ordered (nonempty) set called a *set of worlds* and φ is a function $\varphi\colon W \to \mathbf{BOOL}^{\mathbf{PS}}$ such that for every $u \in W$, the function $\varphi(u)\colon \mathbf{PS} \to \mathbf{BOOL}$ is an assignment of truth values to the propositional symbols in $\mathbf{PS}$ satisfying the following property. For all $u, v \in W$, for all $\mathbf{P}_i \in \mathbf{PS}$,

$$\text{if } u \leq v \text{ and } \varphi(u)(\mathbf{P}_i) = \mathbf{true}, \ \text{ then } \varphi(v)(\mathbf{P}_i) = \mathbf{true}.$$

As we said in our informal comments, truth can't decrease when we move from a world u to a world v with $u \leq v$ but truth can increase; it is possible that $\varphi(u)(\mathbf{P}_i) = \mathbf{false}$ and yet, $\varphi(v)(\mathbf{P}_i) = \mathbf{true}$.

If $W = \{0, 1\}$ ordered so that $0 \leq 1$ and if φ is given by

$$\begin{aligned} \varphi(0)(\mathbf{P}_i) &= \mathbf{false} \\ \varphi(1)(\mathbf{P}_i) &= \mathbf{true}, \end{aligned}$$

then $\mathcal{K}_{\text{bad}} = (W, \varphi)$ is a Kripke structure.

We use Kripke models to define the semantics of propositions as follows.

Definition 2.8. Given a Kripke model $\mathcal{K} = (W, \varphi)$, for every $u \in W$ and for every proposition P we say that P *is satisfied by* $\mathcal{K}$ *at* u and we write $\varphi(u)(P) = \mathbf{true}$ iff

(a) If $P = \mathbf{P}_i \in \mathbf{PS}$, then $\varphi(u)(\mathbf{P}_i) = \mathbf{true}$.

(b) If $P = Q \wedge R$, then $\varphi(u)(Q) = \mathbf{true}$ and $\varphi(u)(R) = \mathbf{true}$.

(c) If $P = Q \vee R$, then $\varphi(u)(Q) = \mathbf{true}$ or $\varphi(u)(R) = \mathbf{true}$.

(d) If $P = Q \Rightarrow R$, then for all v such that $u \leq v$, if $\varphi(v)(Q) = \mathbf{true}$, then $\varphi(v)(R) = \mathbf{true}$.

(e) If $P = \neg Q$, then for all v such that $u \leq v$, $\varphi(v)(Q) = \mathbf{false}$,

(f) $\varphi(u)(\perp) = \mathbf{false}$; that is, $\perp$ is not satisfied by $\mathcal{K}$ at u (for any $\mathcal{K}$ and any u).

We say that P is *valid in* $\mathcal{K}$ (or that $\mathcal{K}$ is a *model* of P) iff P is satisfied by $\mathcal{K} = (W, \varphi)$ at u for all $u \in W$ and we say that P is *intuitionistically valid* iff P is valid in every Kripke model $\mathcal{K}$.

When P is satisfied by $\mathcal{K}$ at u we also say that P *is true at* u *in* $\mathcal{K}$. Note that the truth at $u \in W$ of a proposition of the form $Q \Rightarrow R$ or $\neg Q$ depends on the truth of Q and R at all "future worlds," $v \in W$, with $u \leq v$. Observe that classical truth value semantics corresponds to the special case where W consists of a single element (a single world).

Given the Kripke structure $\mathcal{K}_{\rm bad}$ defined earlier, the reader should check that the proposition $P = (\mathbf{P}_i \vee \neg \mathbf{P}_i)$ has the value **false** at 0 because $\varphi(0)(\mathbf{P}_i) = \mathbf{false}$, but $\varphi(1)(\mathbf{P}_i) = \mathbf{true}$, so clause (e) fails for $\neg \mathbf{P}_i$ at $u = 0$. Therefore, $P = (\mathbf{P}_i \vee \neg \mathbf{P}_i)$ is not valid in $\mathcal{K}_{\rm bad}$ and thus, it is not intuitionistically valid. We escaped the classical truth value semantics by using a universe with two worlds. The reader should also check that

$$\begin{aligned} \varphi(u)(\neg\neg P) = \mathbf{true} \quad \text{iff} \quad & \text{for all } v \text{ such that } u \leq v \\ & \text{there is some } w \text{ with } v \leq w \text{ so that } \varphi(w)(P) = \mathbf{true}. \end{aligned}$$

This shows that in Kripke semantics, $\neg\neg P$ is weaker than P, in the sense that $\varphi(u)(\neg\neg P) = \mathbf{true}$ does not necessarily imply that $\varphi(u)(P) = \mathbf{true}$. The reader should also check that the proposition $\neg\neg \mathbf{P}_i \Rightarrow \mathbf{P}_i$ is not valid in the Kripke structure $\mathcal{K}_{\rm bad}$.

As we said in the previous section, Kripke semantics is a perfect fit to intuitionistic provability in the sense that soundness and completeness hold.

Proposition 2.8. *(Soundness of $\mathcal{N}_i^{\Rightarrow,\wedge,\vee,\perp}$ and $\mathcal{NG}_i^{\Rightarrow,\wedge,\vee,\perp}$) If a proposition P is provable in the system $\mathcal{N}_i^{\Rightarrow,\wedge,\vee,\perp}$ (or $\mathcal{NG}_i^{\Rightarrow,\wedge,\vee,\perp}$), then it is valid in every Kripke model, that is, it is intuitionistically valid.*

Proposition 2.8 is not hard to prove. We consider any deduction of a proposition P from a set of premises Γ and we prove that for every Kripke model $\mathcal{K} = (W, \varphi)$, for every $u \in W$, if every premise in Γ is satisfied by $\mathcal{K}$ at u, then P is also satisfied by $\mathcal{K}$ at u. This is obvious for the axioms and it is easy to see that the inference rules preserve this property.

Completeness also holds, but it is harder to prove (see van Dalen [51]).

Theorem 2.9. *(Completeness of $\mathcal{N}_i^{\Rightarrow,\wedge,\vee,\perp}$ and $\mathcal{NG}_i^{\Rightarrow,\wedge,\vee,\perp}$) If a proposition P is intuitionistically valid, then P is provable in the system $\mathcal{N}_i^{\Rightarrow,\wedge,\vee,\perp}$ (or $\mathcal{NG}_i^{\Rightarrow,\wedge,\vee,\perp}$).*

Another proof of completeness for a different proof system for propositional intuitionistic logic (a Gentzen-sequent calculus equivalent to $\mathcal{NG}_i^{\Rightarrow,\wedge,\vee,\perp}$) is given in Takeuti [49]. We find this proof more instructive than van Dalen's proof. This proof also shows that if a

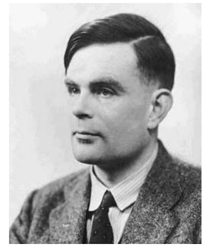


Figure 2.4: Alonzo Church, 1903–1995 (left) and Alan Turing, 1912–1954 (right)

proposition P is not intuitionistically provable, then there is a Kripke model $\mathcal{K}$ where W is a *finite tree* in which P is not valid. Such a Kripke model is called a *counterexample* for P.

Several times in this chapter, we have claimed that certain formulae are not provable in some logical system. What kind of reasoning do we use to validate such claims? In the next section, we briefly address this question as well as related ones.

2.12 Decision Procedures, Proof Normalization

In the previous sections we saw how the rules of mathematical reasoning can be formalized in various natural deduction systems and we defined a precise notion of proof. We observed that finding a proof for a given proposition was not a simple matter, nor was it to acertain that a proposition is unprovable. Thus, it is natural to ask the following question.

The Decision Problem: Is there a general procedure that takes any arbitrary proposition P as input, always terminates in a finite number of steps, and tells us whether P is provable?

Clearly, it would be very nice if such a procedure existed, especially if it also produced a proof of P when P is provable.

Unfortunately, for rich enough languages, such as first-order logic (discussed in Section 2.15) it is impossible to find such a procedure. This deep result known as the *undecidability of the decision problem* or *Church's theorem* was proven by A. Church in 1936 (actually, Church proved the undecidability of the validity problem but, by Gödel's completeness theorem, validity and provability are equivalent). We will present a version of Church's theorem in Section 9.4.

Proving Church's theorem is hard and a lot of work. One needs to develop a good deal of what is called the *theory of computation*. This involves defining models of computation such as *Turing machines* and proving other deep results such as the *undecidability of the halting problem* and the *undecidability of the Post correspondence problem*, among other things. We will discuss these topics in Chapters 3, 4, 5, 7, 8 and 9. See also Hopcroft, Motwani, and Ullman [29] and Lewis and Papadimitriou [35].

So our hopes to find a "universal theorem prover" are crushed. However, if we restrict ourselves to propositional logic, classical or intuitionistic, it turns out that procedures solving the decision problem do exist and they even produce a proof of the input proposition when that proposition is provable.

Unfortunately, proving that such procedures exist, and are correct in the propositional case is rather difficult, especially for intuitionistic logic. The difficulties have a lot to do with our choice of a natural deduction system. Indeed, even for the system $\mathcal{N}_m^{\Rightarrow}$ (or $\mathcal{NG}_m^{\Rightarrow}$), provable propositions may have infinitely many proofs. This makes the search process impossible; when do we know how to stop, especially if a proposition is not provable. The problem is that proofs may contain redundancies (Gentzen said "detours"). A typical example of redundancy is when an elimination immediately follows an introduction, as in the following example:

$$\dfrac{\dfrac{\dfrac{\dfrac{y\colon ((R \Rightarrow R) \Rightarrow Q) \to ((R \Rightarrow R) \Rightarrow Q) \quad x\colon (R \Rightarrow R) \to (R \Rightarrow R)}{x\colon (R \Rightarrow R), y\colon ((R \Rightarrow R) \Rightarrow Q) \to Q}}{x\colon (R \Rightarrow R) \to ((R \Rightarrow R) \Rightarrow Q) \Rightarrow Q}\ y}{\to (R \Rightarrow R) \Rightarrow (((R \Rightarrow R) \Rightarrow Q) \Rightarrow Q)}\ x \qquad \dfrac{z\colon R \to R}{\to R \Rightarrow R}\ z}{\to ((R \Rightarrow R) \Rightarrow Q) \Rightarrow Q}$$

The blue deduction already has $((R \Rightarrow R) \Rightarrow Q) \Rightarrow Q$ as conclusion but it is not a proof because the assumption $x\colon (R \Rightarrow R)$ is present. However we have a proof of $R \Rightarrow R$, namely

$$\dfrac{z\colon R \to R}{\to R \Rightarrow R}\ z$$

We can obtain a proof of $((R \Rightarrow R) \Rightarrow Q) \Rightarrow Q$ from the blue deduction tree by replacing the leaf labeled $x\colon (R \Rightarrow R) \to (R \Rightarrow R)$ by the proof tree for $R \Rightarrow R$, obtaining

$$\dfrac{\dfrac{y\colon ((R \Rightarrow R) \Rightarrow Q) \to ((R \Rightarrow R) \Rightarrow Q) \qquad \dfrac{z\colon R \to R}{\to R \Rightarrow R}\ z}{x\colon (R \Rightarrow R), y\colon ((R \Rightarrow R) \Rightarrow Q) \to Q}}{x\colon (R \Rightarrow R) \to ((R \Rightarrow R) \Rightarrow Q) \Rightarrow Q}\ y$$

The above is not quite a proof tree, but it becomes one if we delete the premise $x\colon (R \Rightarrow R)$ which is now redundant:

$$\dfrac{\dfrac{y\colon ((R \Rightarrow R) \Rightarrow Q) \to ((R \Rightarrow R) \Rightarrow Q) \qquad \dfrac{z\colon R \to R}{\to R \Rightarrow R}\ z}{y\colon ((R \Rightarrow R) \Rightarrow Q) \to Q}}{\to ((R \Rightarrow R) \Rightarrow Q) \Rightarrow Q}\ y$$

The procedure that we just described for eliminating a redundancy can be generalized. Consider the deduction tree below in which $\mathcal{D}_1$ denotes a deduction with conclusion $\Gamma, x\colon A \to B$ and $\mathcal{D}_2$ denotes a deduction with conclusion $\Delta \to A$.

$$\dfrac{\dfrac{\begin{array}{c}\mathcal{D}_1\\ \Gamma, x\colon A \to B\end{array}}{\Gamma \to A \Rightarrow B} \qquad \begin{array}{c}\mathcal{D}_2\\ \Delta \to A\end{array}}{\Gamma \cup \Delta \to B}$$

It should be possible to construct a deduction for $\Gamma \to B$ from the two deductions $\mathcal{D}_1$ and $\mathcal{D}_2$ without using at all the hypothesis $x\colon A$. This is indeed the case. If we look closely at the deduction $\mathcal{D}_1$, from the shape of the inference rules, assumptions are never created, and the leaves must be labeled with expressions of the form either

(1) $\Gamma, \Lambda, x\colon A \to A$, or

(2) $\Gamma', \Lambda, x\colon A, y\colon C \to C$ if $\Gamma = \Gamma', y\colon C$ and $y \neq x$, or

(3) $\Gamma, \Lambda, x\colon A, y\colon C \to C$ if $y\colon C \notin \Gamma$ and $y \neq x$.

We can form a new deduction for $\Gamma \to B$ as follows. In $\mathcal{D}_1$, wherever a leaf of the form $\Gamma, \Lambda, x\colon A \to A$ occurs, replace it by the deduction obtained from $\mathcal{D}_2$ by adding Λ to the premise of each sequent in $\mathcal{D}_2$.

In our previous example, we have $A = (R \Rightarrow R)$, $B = ((R \Rightarrow R) \Rightarrow Q) \Rightarrow Q$, $C = (R \Rightarrow R) \Rightarrow Q$, $\Gamma = \Delta = \Lambda = \emptyset$.

Actually, one should be careful to first make a fresh copy of $\mathcal{D}_2$ by renaming all the variables so that clashes with variables in $\mathcal{D}_1$ are avoided. Finally, delete the assumption $x\colon A$ from the premise of every sequent in the resulting proof. The resulting deduction is obtained by a kind of substitution and may be denoted as $\mathcal{D}_1[\mathcal{D}_2/x]$, with some minor abuse of notation. Note that the assumptions $x\colon A$ occurring in the leaves of type (2) or (3) were never used anyway. The step that consists in transforming the above redundant proof figure into the deduction $\mathcal{D}_1[\mathcal{D}_2/x]$ is called a *reduction step* or *normalization step*.

The idea of *proof normalization* goes back to Gentzen ([20], 1935). Gentzen noted that (formal) proofs can contain redundancies, or "detours," and that most complications in the analysis of proofs are due to these redundancies. Thus, Gentzen had the idea that the analysis of proofs would be simplified if it were possible to show that every proof can be converted to an equivalent irredundant proof, a proof in *normal form*. Gentzen proved a technical result to that effect, the "cut-elimination theorem," for a sequent-calculus formulation of first-order logic [20]. Cut-free proofs are direct, in the sense that they never use auxiliary lemmas via the cut rule.

Remark: It is important to note that Gentzen's result gives a particular algorithm to produce a proof in normal form. Thus we know that every proof can be reduced to some normal form using a specific strategy, but there may be more than one normal form, and certain normalization strategies may not terminate.

About 30 years later, Prawitz ([41], 1965) reconsidered the issue of proof normalization, but in the framework of natural deduction rather than the framework of sequent calculi.[1]

[1]This is somewhat ironical, inasmuch as Gentzen began his investigations using a natural deduction system, but decided to switch to sequent calculi (known as Gentzen systems) for technical reasons.

Figure 2.5: Haskell B. Curry, 1900–1982

Prawitz explained very clearly what redundancies are in systems of natural deduction, and he proved that every proof can be reduced to a normal form. Furthermore, this normal form is *unique*. A few years later, Prawitz ([42], 1971) showed that in fact, every reduction sequence terminates, a property also called *strong normalization*.

A remarkable connection between proof normalization and the notion of computation must also be mentioned. Curry (1958) made the remarkably insightful observation that certain typed combinators can be viewed as representations of proofs (in a Hilbert system) of certain propositions. (See in Curry and Feys [7] (1958), Chapter 9E, pages 312–315.)

Building up on this observation, Howard ([30], 1969) described a general correspondence among propositions and types, proofs in natural deduction and certain typed λ-terms, and proof normalization and β-reduction (The simply typed λ-calculus was invented by Church, 1940). This correspondence, usually referred to as the *Curry–Howard isomorphism* or *formulae-as-types principle*, is fundamental and very fruitful.

Let us elaborate on this correspondence.

2.13 The Simply-Typed λ-Calculus

First we need to define the simply-typed λ-calculus and the first step is to define simple types. We assume that we have a countable set $\{\mathbf{T}_0, \mathbf{T}_1, \ldots, \mathbf{T}_n, \ldots\}$ of *base types* (or *atomic types*). For example, the base types may include types such as Nat for the natural numbers, Bool for the booleans, String for strings, Tree for trees, *etc.* In the Curry–Howard isomorphism, the base types correspond to the propositional symbols $\{\mathbf{P}_0, \mathbf{P}_1, \ldots, \mathbf{P}_n, \ldots\}$.

Definition 2.9. The *simple types* σ are defined inductively as follows:

(1) If $\mathbf{T}_i$ is a base type, then $\mathbf{T}_i$ is a simple type.

(2) If σ and τ are simple types, then $(\sigma \to \tau)$ is a simple type.

Thus $(\mathbf{T}_1 \to \mathbf{T}_1)$, $(\mathbf{T}_1 \to (\mathbf{T}_2 \to \mathbf{T}_1))$ $((\mathbf{T}_1 \to \mathbf{T}_2) \to \mathbf{T}_1)$, are simple types.

The standard abbreviation for $(\sigma_1 \to (\sigma_2 \to (\cdots \to \sigma_n)))$ is $\sigma_1 \to \sigma_2 \to \cdots \to \sigma_n$.

There is obviously a bijection between propositions and simple types. Every propositional symbol $\mathbf{P}_i$ can be viewed as a base type, and the proposition $(P \Rightarrow Q)$ corresponds to the simple type $(P \to Q)$. The only difference is that the custom is to use $\Rightarrow$ to denote logical implication and $\to$ for simple types. The reason is that intuitively a simple type $(\sigma \to \tau)$ corresponds to a *set of functions* from a domain of type σ to a range of type τ.

The next crucial step is to define simply-typed λ-terms. This is done in two stages. First we define *raw simply-typed λ-terms*. They have a simple inductive definition but they do not necessarily type-check so we define some type-checking rules that turn out to be *the Gentzen-style deduction proof rules annotated with simply-typed λ-terms*. These simply-typed λ-terms are representations of natural deductions.

We have a countable set of variables $\{x_0, x_1, \ldots, x_n \ldots\}$ that correspond to the atomic raw λ-terms. These are also the variables that are used for tagging assumptions when constructing deductions.

Definition 2.10. The *raw simply-typed λ-terms* (for short *raw terms* or *λ-terms*) M are defined inductively as follows:

(1) If x_i is a variable, then x_i is a raw term.

(2) If M and N are raw terms, then (MN) is a raw term called an *application*.

(3) If M is a raw term, σ is a simple type, and x is a variable, then the expression $\lambda x\colon \sigma.\, M$ is a raw term called a *λ-abstraction*.

Matching parentheses may be dropped or added for convenience. In a raw λ-term M, a variable x appearing in an expression $\lambda x\colon \sigma$ is said to be *bound* in M. The other variables in M (if any) are said to be *free* in M. A λ-term M is *closed* if it has no free variables.

For example, in the term $\lambda x\colon \sigma.\,(yx)$, the variable x is bound and the variable y is free. This term is not closed. The term $\lambda y\colon \sigma \to \sigma.\,(\lambda x\colon \sigma.\,(yx))$ is closed.

The intuition is that a term of the form $\lambda x\colon \sigma.\, M$ represents a function. How such a function operates will be defined in terms of β-reduction.

Definition 2.11. The *depth* $d(M)$ of a raw λ-term M is defined inductively as follows.

1. If M is a variable x, then $d(x) = 0$.

2. If M is an application $(M_1 M_2)$, then $d(M) = \max\{d(M_1), d(M_2)\} + 1$.

3. If M is a λ-abstraction $(\lambda x\colon \sigma.\, M_1)$, then $d(M) = d(M_1) + 1$.

It is pretty clear that raw λ-terms have representations as (ordered) labeled trees.

Definition 2.12. Given a raw λ-term M, the *tree* $\mathrm{tree}(M)$ representing M is defined inductively as follows:

1. If M is a variable x, then $\mathrm{tree}(M)$ is the one-node tree labeled x.

2. If M is an application $(M_1 M_2)$, then $\mathrm{tree}(M)$ is the tree with a binary root node labeled $.$, and with a left subtree $\mathrm{tree}(M_1)$ and a right subtree $\mathrm{tree}(M_2)$.

3. If M is a λ-abstraction $\lambda x\colon \sigma.\, M_1$, then $\text{tree}(M)$ is the tree with a unary root node labeled $\lambda x\colon \sigma$, and with one subtree $\text{tree}(M_1)$.

Definition 2.12 is illustrated in Figure 2.6.

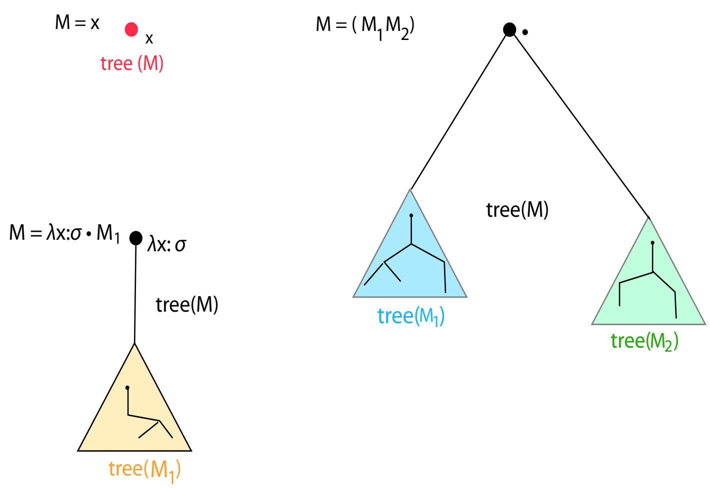


Figure 2.6: The tree $\text{tree}(M)$ associated with a raw λ-term M.

Obviously, the depth $d(M)$ of raw λ-term is the depth of its tree representation $\text{tree}(M)$.

Definition 2.12 could be used to deal with bound variables. For every leaf labeled with a bound variable x, we draw a backpointer to an ancestor of x determined as follows. Given a leaf labeled with a bound variable x, climb up to the closest ancestor labeled $\lambda x\colon \sigma$, and draw a backpointer to this node. Then all bound variables can be erased. See Figure 2.7 for an example.

Definition 2.10 allows the construction of undesirable terms such as (xx) or $(\lambda x\colon \sigma.\,(xx))(\lambda x\colon \sigma.\,(xx))$ because no type-checking is done. Part of the problem is that the variables occurring in a raw term have not been assigned types. This can be done using a *context* (or *type assignment*), which is a set of pairs $\Gamma = \{x_1\colon \sigma_1, \ldots, x_n\colon \sigma_n\}$ where the σ_i are simple types. Once a type assignment has been provided, the type-checking rules are basically the proof rules of natural deduction in Gentzen-style. The fact that a raw term M has type σ given a type assignment Γ that assigns types to all the free variables in M is written as

$$\Gamma \triangleright M\colon \sigma.$$

Such an expression is called a *judgement*. The symbol $\triangleright$ is used instead of the symbol $\rightarrow$ because $\rightarrow$ occurs in simple types. Here are the typing-checking rules.

Definition 2.13. The *type-checking rules* of the simply-typed λ-calculus $\lambda^{\rightarrow}$ are listed below:

$$\Gamma, x\colon \sigma \triangleright x\colon \sigma$$

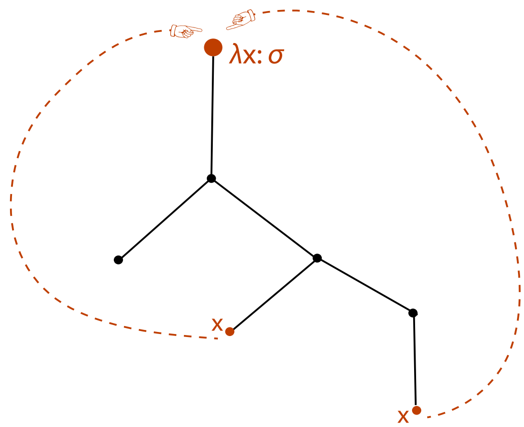


Figure 2.7: Using backpointers to deal with bound variables.

$$\frac{\Gamma, x\colon \sigma \triangleright M\colon \tau}{\Gamma \triangleright (\lambda x\colon \sigma.\, M)\colon \sigma \to \tau} \quad (\textit{abstraction})$$

$$\frac{\Gamma \triangleright M\colon \sigma \to \tau \quad \Delta \triangleright N\colon \tau}{\Gamma \cup \Delta \triangleright (MN)\colon \tau} \quad (\textit{application})$$

We write $\vdash \Gamma \triangleright M\colon \sigma$ to express that the judgement $\Gamma \triangleright M\colon \sigma$ is provable. Given a raw simply-typed λ-term M, if there is a type-assigment Γ and a simple type σ such that the judgement $\Gamma \triangleright M\colon \sigma$ is provable, we say that M *type-checks with type* σ.

It can be shown by induction on the depth of raw terms that for a fixed type-assigment Γ, if a raw simply-typed λ-term M type-checks with some simple type σ, then σ is unique.

The correspondence between proofs in natural deduction and simply-typed λ-terms (the Curry/Howard isomorphism) is now clear: the blue term is a *representation of the deduction* of the sequents $\Gamma, x\colon \sigma \to \sigma$, $\Gamma \to \sigma \Rightarrow \tau$, and $\Gamma \cup \Delta \to \tau$, with the types $\sigma, \sigma \Rightarrow \tau$ and τ viewed as propositions. Note that proofs correspond to closed λ-terms.

For example, we have the type-checking proof

$$\dfrac{\dfrac{y\colon ((R \Rightarrow R) \Rightarrow Q) \triangleright y\colon ((R \Rightarrow R) \Rightarrow Q) \qquad \dfrac{z\colon R \triangleright z\colon R}{\triangleright \lambda z\colon R.\, z\colon R \Rightarrow R}}{y\colon ((R \Rightarrow R) \Rightarrow Q) \triangleright y(\lambda z\colon R.\, z)\colon Q}}{\triangleright \lambda y\colon ((R \Rightarrow R) \Rightarrow Q).\, y(\lambda z\colon R.\, z)\colon ((R \Rightarrow R) \Rightarrow Q) \Rightarrow Q}$$

which shows that the simply-typed λ-term

$$M = \lambda y\colon ((R \Rightarrow R) \Rightarrow Q).\, y(\lambda z\colon R.\, z)$$

represents the proof

$$\dfrac{\dfrac{y\colon ((R \Rightarrow R) \Rightarrow Q) \to ((R \Rightarrow R) \Rightarrow Q) \qquad \dfrac{z\colon R \to R}{\to R \Rightarrow R}}{y\colon ((R \Rightarrow R) \Rightarrow Q) \to Q}}{\to ((R \Rightarrow R) \Rightarrow Q) \Rightarrow Q}$$

The proposition $((R \Rightarrow R) \Rightarrow Q) \Rightarrow Q$ being proven is the type of the λ-term M. The tree representing the λ-term $M = \lambda y\colon ((R \Rightarrow R) \Rightarrow Q).\, y(\lambda z\colon R.\, z)$ is shown in Figure 2.8.

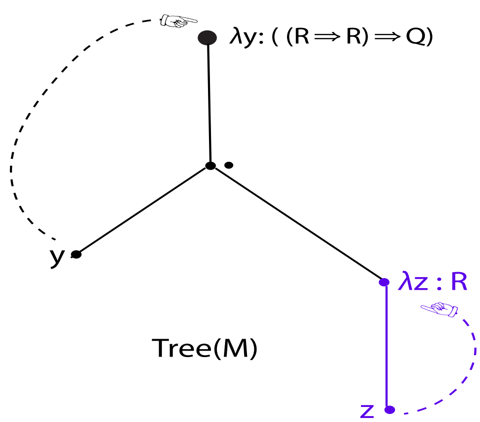


Figure 2.8: The tree representation of the λ-term M.

Furthermore, and this is the deepest aspect of the Curry/Howard isomorphism, proof normalization corresponds to β-reduction in the simply-typed λ-calculus.

The notion of β-reduction is defined in terms of substitutions. A *substitution* φ is a finite set of pairs $\varphi = \{(x_1, N_1), \ldots, (x_n, N_n)\}$, where the x_i are distinct variables and the N_i are raw λ-terms. We write

$$\varphi = [N_1/x_1, \ldots, N_n/x_n] \quad \text{or} \quad \varphi = [x_1 := N_1, \ldots, x_n := N_n].$$

The second notation indicates more clearly that each term N_i is substituted for the variable x_i and it seems to have been almost universally adopted.

Given a substitution $\varphi = [x_1 := N_1, \ldots, x_n := N_n]$, for any variable x_i, we denote by φ_{-x_i} the new substitution where the pair (x_i, N_i) is replaced by the pair (x_i, x_i) (that is, the new substitution leaves x_i unchanged).

Given any raw λ-term M and any substitution $\varphi = [x_1 := N_1, \ldots, x_n := N_n]$, we define the raw λ-term $M[\varphi]$, the result of applying the substitution φ to M, as follows:

(1) If $M = y$, with $y \neq x_i$ for $i = 1, \ldots, n$, then $M[\varphi] = y = M$.

(2) If $M = x_i$ for some $i \in \{1, \ldots, n\}$, then $M[\varphi] = N_i$.

(3) If $M = (PQ)$, then $M[\varphi] = (P[\varphi]Q[\varphi])$.

(4) If $M = \lambda x\colon \sigma.\, N$ and $x \neq x_i$ for $i = 1, \ldots, n$, then $M[\varphi] = \lambda x\colon \sigma.\, N[\varphi]$,

(5) If $M = \lambda x\colon \sigma.\, N$ and $x = x_i$ for some $i \in \{1, \ldots, n\}$, then $M[\varphi] = \lambda x\colon \sigma.\, N[\varphi]_{-x_i}$.

There is a problem with the present definition of a substitution in Cases (4) and (5), which is that the result of substituting a term N_i containing the variable x free causes this variable to become bound after the substitution. We say that x is *captured*. To remedy this problem, Church defined *α-conversion*.

The idea of α-conversion is that in a raw term M any subterm of the form $\lambda x\colon \sigma.\, P$ can be replaced by the subterm $\lambda z\colon \sigma.\, P[x := z]$ where z is a new variable not occurring at all (free or bound) in M to obtain a new term M'. We write $M \equiv_\alpha M'$ and we view M and M' as equivalent.

For example, $\lambda x\colon \sigma.\, yx \equiv_\alpha \lambda z\colon \sigma.\, yz$ and

$$\lambda y\colon \sigma \to \sigma.\, (\lambda x\colon \sigma.\, yx) \equiv_\alpha \lambda w\colon \sigma \to \sigma.\, (\lambda z\colon \sigma.\, wz).$$

The variables x and y are just place-holders.

Then given a raw λ-term M and a substitution $\varphi = [x_1 := N_1, \ldots, x_n := N_n]$, before applying φ to M we first apply some α-conversion to rename all bound variables in M obtaining $M' \equiv_\alpha M$ so that they do not occur in any of the N_i, and then safely apply the substitution φ to M' without any capture of variables. We say that the term M' is *safe* for the substitution φ. The details are a bit tedious and we omit them. We refer the interested reader to Gallier [17] for a comprehensive discussion.

The following result shows that substitutions behave well with respect to type-checking. Given a context $\Gamma = \{x_1\colon \sigma_1, \ldots, x_n\colon \sigma_n\}$, we let $\Gamma(x_i) = \sigma_i$.

Proposition 2.10. *For any raw λ-term M and any substitution $\varphi = [x_1 := N_1, \ldots, x_n := N_n]$, whose domain contains the set of free variables of M, if the judgement $\Gamma \rhd M\colon \tau$ is provable for some context Γ and some simple type τ, and if there is some context Δ such that for every free variable x_j in M the judgement $\Delta \rhd N_j\colon \Gamma(x)$ is provable, then there some $M' \equiv_\alpha M$ such that the judgment $\Delta \rhd M'[\varphi]\colon \tau$ is provable.*

Finally we define β-reduction and β-conversion as follows.

Definition 2.14. The relation $\longrightarrow_\beta$, called *immediate β-reduction*, is the smallest relation satisfying the following properties for all raw λ-terms M, N, P, Q:

$$(\lambda x\colon \sigma.\, M)N \longrightarrow_\beta M[x := N]$$

provided that M is safe for $[x := N]$;

$$\frac{M \longrightarrow_\beta N}{MQ \longrightarrow_\beta NQ} \qquad \frac{M \longrightarrow_\beta N}{PM \longrightarrow_\beta PN} \qquad \text{for all } P, Q \qquad (\textit{congruence})$$

$$\frac{M \longrightarrow_\beta N}{\lambda x\colon \sigma.\, M \longrightarrow_\beta \lambda x\colon \sigma.\, N} \qquad \text{for all } \sigma \tag{ξ}$$

The transitive closure of $\longrightarrow_\beta$ is denoted by $\stackrel{+}{\longrightarrow}_\beta$, the reflexive and transitive closure of $\longrightarrow_\beta$ is denoted by $\stackrel{*}{\longrightarrow}_\beta$, and we define *$\beta$-conversion*, denoted by $\stackrel{*}{\longleftrightarrow}_\beta$, as the smallest equivalence relation $\stackrel{*}{\longleftrightarrow}_\beta = (\longrightarrow_\beta \cup \longrightarrow_\beta^{-1})^*$ containing $\longrightarrow_\beta$.

For example, we have

$$\begin{aligned}
&(\lambda u\colon \sigma.\, (vu))\big((\lambda x\colon \sigma \to \sigma.\, (xy))(\lambda z\colon \sigma.\, z)\big) \longrightarrow_\beta \\
&\qquad (\lambda u\colon \sigma.\, (vu))(\lambda x\colon \sigma \to \sigma.\, (xy))[x := (\lambda z\colon \sigma.\, z)] = (\lambda u\colon \sigma.\, (vu))\big((\lambda z\colon \sigma.\, z)y\big) \\
&\qquad\qquad \longrightarrow_\beta (\lambda u\colon \sigma.\, (vu))z[z := y] = (\lambda u\colon \sigma.\, (vu))y \longrightarrow_\beta (vu)[u := y] = vy.
\end{aligned}$$

The following result shows that β-reduction (and β-conversion) behave well with respect to type-checking.

Proposition 2.11. *For any two raw λ-terms M and N, if there is a proof of the judgement $\Gamma \triangleright M\colon \sigma$ for some context Γ and some simple type σ, and if $M \stackrel{+}{\longrightarrow}_\beta N$ (or $M \stackrel{*}{\longleftrightarrow}_\beta N$), then the judgement $\Gamma \triangleright N\colon \sigma$ is provable. Thus β-reduction and β-conversion preserve type-checking.*

We say that a λ-term M is *β-irreducible* or a *β-normal form* if there is *no* term N such that $M \longrightarrow_\beta N$.

The fundamental result about the simply-typed λ-calculus is this.

Theorem 2.12. *For every raw λ-term M, if M type-checks, which means that there a provable judgement $\Gamma \triangleright M\colon \sigma$ for some context Γ and some simple type σ, then the following results hold:*

(1) If $M \stackrel{*}{\longrightarrow}_\beta M_1$ and $M \stackrel{*}{\longrightarrow}_\beta M_2$, then there is some M_3 such that $M_1 \stackrel{*}{\longrightarrow}_\beta M_3$ and $M_2 \stackrel{*}{\longrightarrow}_\beta M_3$. We say that $\stackrel{*}{\longrightarrow}_\beta$ is confluent.

(2) Every reduction sequence $M \stackrel{+}{\longrightarrow}_\beta N$ is finite. We that that the simply-typed λ-calculus is strongly normalizing (for short, SN).

As a consequence of (1) and (2), there is a unique β-irreducible term N (called a β-normal form) such that $M \stackrel{*}{\longrightarrow}_\beta N$.

A proof of Theorem 2.12 can be found in Gallier [15]. See also Gallier [17] which contains a thorough discussion of the techniques involved in proving these results.

In Theorem 2.12, the fact that the term M type-checks is crucial. Indeed the term

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

which does not type-check (we omitted the type tags σ of the variable x since they do not play any role), gives rise to an infinite β-reduction sequence!

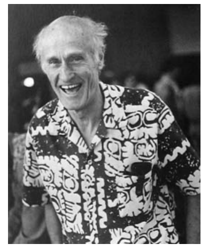


Figure 2.9: Stephen C. Kleene, 1909–1994

In summary, the correspondence between proofs in intuitionistic logic and typed λ-terms on one hand and between proof normalization and β-reduction, can be used to translate results about typed λ-terms into results about proofs in intuitionistic logic. These results can be generalized to typed λ-calculi with product types and union types; see Gallier [15].

Using some suitable intuitionistic sequent calculi and Gentzen's cut elimination theorem or some suitable typed λ-calculi and (strong) normalization results about them, it is possible to prove that there is a decision procedure for propositional intuitionistic logic. However, it can also be shown that the time-complexity of any such procedure is very high. As a matter of fact, it was shown by Statman (1979) that deciding whether a proposition is intuitionisticaly provable is P-space complete; see [46] and Section 13.4. Here, we are alluding to *complexity theory*, another active area of computer science, see Hopcroft, Motwani, and Ullman [29] and Lewis and Papadimitriou [35].

Readers who wish to learn more about these topics can read my two survey papers Gallier [15] (On the Correspondence Between Proofs and λ-Terms) and Gallier [14] (A Tutorial on Proof Systems and Typed λ-Calculi), both available on the website
http://www.cis.upenn.edu/~jean/gbooks/logic.html and the excellent introduction to proof theory by Troelstra and Schwichtenberg [50].

Anybody who really wants to understand logic should of course take a look at Kleene [32] (the famous "I.M."), but this is not recommended to beginners.

2.14 Completeness and Counter-Examples

Let us return to the question of deciding whether a proposition is not provable. To simplify the discussion, let us restrict our attention to propositional classical logic. So far, we have presented a very *proof-theoretic* view of logic, that is, a view based on the notion of provability as opposed to a more *semantic* view of based on the notions of truth and models. A possible excuse for our bias is that, as Peter Andrews (from CMU) puts it, "truth is elusive." Therefore, it is simpler to understand what truth is in terms of the more "mechanical" notion of provability. (Peter Andrews even gave the subtitle

To Truth Through Proof

to his logic book Andrews [1].)

Figure 2.10: Peter Andrews, 1937–

However, mathematicians are not mechanical theorem provers (even if they prove lots of stuff). Indeed, mathematicians almost always think of the objects they deal with (functions, curves, surfaces, groups, rings, etc.) as rather concrete objects (even if they may not seem concrete to the uninitiated) and not as abstract entities solely characterized by arcane axioms.

It is indeed natural and fruitful to try to interpret formal statements semantically. For propositional classical logic, this can be done quite easily if we interpret atomic propositional letters using the truth values **true** and **false**, as explained in Section 2.10. Then, the crucial point that *every provable proposition (say in $\mathcal{NG}_c^{\Rightarrow,\vee,\wedge,\perp}$) has the value* **true** *no matter how we assign truth values to the letters in our proposition*. In this case, we say that P is *valid*.

The fact that provability implies validity is called *soundness* or *consistency* of the proof system. The soundness of the proof system $\mathcal{NG}_c^{\Rightarrow,\vee,\wedge,\perp}$ is easy to prove, as sketched in Section 2.10.

We now have a method to show that a proposition P is not provable: find some truth assignment that makes P **false**.

Such an assignment falsifying P is called a *counterexample*. If P has a counterexample, then it can't be provable because if it were, then by soundness it would be **true** for all possible truth assignments.

But now, another question comes up. If a proposition is not provable, can we always find a counterexample for it? Equivalently, *is every valid proposition provable*? If every valid proposition is provable, we say that our proof system is *complete* (this is the *completeness* of our system).

The system $\mathcal{NG}_c^{\Rightarrow,\vee,\wedge,\perp}$ is indeed complete. In fact, *all* the classical systems that we have discussed are sound and complete. Completeness is usually a lot harder to prove than soundness. For first-order classical logic, this is known as *Gödel's completeness theorem* (1929). Again, we refer our readers to Gallier [19], van Dalen [51], or Huth and Ryan [31] for a thorough discussion of these matters. In the first-order case, one has to define *first-order structures* (or *first-order models*).

What about intuitionistic logic?

Well, one has to come up with a richer notion of semantics because it is no longer true that if a proposition is valid (in the sense of our two-valued semantics using **true**, **false**), then it is provable. Several semantics have been given for intuitionistic logic. In our opinion, the most natural is the notion of the *Kripke model*, presented in Section 2.11. Then, again, soundness and completeness hold for intuitionistic proof systems, even in the first-order case

Figure 2.11: Jean-Yves Girard, 1947–

(see Section 2.11 and van Dalen [51]).

In summary, semantic models can be used to provide *counterexamples* of unprovable propositions. This is a quick method to establish that a proposition is not provable.

We close this section by repeating something we said earlier: there isn't just one logic but instead, *many* logics. In addition to classical and intuitionistic logic (propositional and first-order), there are: modal logics, higher-order logics, and *linear logic*, a logic due to Jean-Yves Girard, attempting to unify classical and intuitionistic logic (among other goals).

An excellent introduction to these logics can be found in Troelstra and Schwichtenberg [50]. We warn our readers that most presentations of linear logic are (very) difficult to follow. This is definitely true of Girard's seminal paper [24]. A more approachable version can be found in Girard, Lafont, and Taylor [21], but most readers will still wonder what hit them when they attempt to read it.

In computer science, there is also *dynamic logic*, used to prove properties of programs and *temporal logic* and its variants (originally invented by A. Pnueli), to prove properties of real-time systems. So logic is alive and well.

We now add quantifiers to our language and give the corresponding inference rules.

2.15 Adding Quantifiers; Proof Systems $\mathcal{N}_c^{\Rightarrow,\wedge,\vee,\forall,\exists,\perp}$ and $\mathcal{NG}_c^{\Rightarrow,\wedge,\vee,\forall,\exists,\perp}$

As we mentioned in Section 2.1, atomic propositions may contain variables. The intention is that such variables correspond to arbitrary objects. An example is

$$\text{human}(x) \Rightarrow \text{needs-to-drink}(x).$$

Now in mathematics, we usually prove universal statements, that is statements that hold for all possible "objects," or existential statements, that is, statements asserting the existence of some object satisfying a given property. As we saw earlier, we assert that every human needs to drink by writing the proposition

$$\forall x(\text{human}(x) \Rightarrow \text{needs-to-drink}(x)).$$

Observe that once the quantifier $\forall$ (pronounced "for all" or "for every") is applied to the variable x, the variable x becomes a placeholder and replacing x by y or any other variable *does not change anything*. What matters is the locations to which the outer x points in the inner proposition. We say that x is a *bound variable* (sometimes a "dummy variable").

If we want to assert that some human needs to drink we write

$$\exists x(\text{human}(x) \Rightarrow \text{needs-to-drink}(x));$$

Again, once the quantifier $\exists$ (pronounced "there exists") is applied to the variable x, the variable x becomes a placeholder. However, the intended meaning of the second proposition is very different and weaker than the first. It only asserts the existence of some object satisfying the statement

$$\text{human}(x) \Rightarrow \text{needs-to-drink}(x).$$

Statements may contain variables that are not bound by quantifiers. For example, in

$$\exists x\, \text{parent}(x, y)$$

the variable x is bound but the variable y is not. Here the intended meaning of $\text{parent}(x, y)$ is that x is a parent of y, and the intended meaning of $\exists x\, \text{parent}(x, y)$ is that any given y has some parent x. Variables that are not bound are called *free*. The proposition

$$\forall y \exists x\, \text{parent}(x, y),$$

which contains only bound variables is meant to assert that every y has some parent x. Typically, in mathematics, we only prove statements without free variables. However, statements with free variables may occur during intermediate stages of a proof.

The intuitive meaning of the statement $\forall x P$ is that P holds for all possible objects x, and the intuitive meaning of the statement $\exists x P$ is that P holds for some object x. Thus, we see that it would be useful to use symbols to denote various objects. For example, if we want to assert some facts about the "parent" predicate, we may want to introduce some *constant symbols* (for short, constants) such as "Jean," "Mia," and so on and write

$$\text{parent}(\text{Jean}, \text{Mia})$$

to assert that Jean is a parent of Mia. Often, we also have to use *function symbols* (or *operators, constructors*), for instance, to write a statement about numbers: $+$, $*$, and so on. Using constant symbols, function symbols, and variables, we can form *terms*, such as

$$(x * x + 1) * (3 * y + 2).$$

In addition to function symbols, we also use *predicate symbols*, which are names for atomic properties. We have already seen several examples of predicate symbols: "human," "parent." So, in general, when we try to prove properties of certain classes of objects (people, numbers, strings, graphs, and so on), we assume that we have a certain *alphabet* consisting of constant symbols, function symbols, and predicate symbols. Using these symbols and an infinite supply of variables (assumed distinct from the variables we use to label premises) we can form *terms* and *predicate terms*. We say that we have a *(logical) language*. Using this language, we can write compound statements.

Let us be a little more precise. In a *first-order language* **L** in addition to the logical connectives $\Rightarrow, \wedge, \vee, \neg, \perp, \forall$, and $\exists$, we have a set **L** of *nonlogical symbols* consisting of

(i) A set **CS** of *constant symbols*, $c_1, c_2, \ldots,$.

(ii) A set **FS** of *function symbols*, $f_1, f_2, \ldots,$. Each function symbol f has a *rank* $n_f \geq 1$, which is the number of arguments of f.

(iii) A set **PS** of *predicate symbols*, $P_1, P_2, \ldots,$. Each predicate symbol P has a *rank* $n_P \geq 0$, which is the number of arguments of P. Predicate symbols of rank 0 are *propositional symbols* as in earlier sections.

(iv) The *equality predicate* $=$ is added to our language when we want to deal with equations.

(v) First-order variables $t_1, t_2, \ldots$ used to form *quantified formulae*.

The difference between function symbols and predicate symbols is that function symbols are interpreted as functions defined on a structure (e.g., addition, $+$, on $\mathbb{N}$), whereas predicate symbols are interpreted as properties of objects, that is, they take the value **true** or **false**.

An example is the language of *Peano arithmetic*, $\mathbf{L} = \{0, S, +, *, =\}$, where 0 is a constant symbol, S is a function symbol with one argument, and $+, *$ are function symbols with two arguments. Here, the intended structure is $\mathbb{N}$, 0 is of course zero, S is interpreted as the function $S(n) = n + 1$, the symbol $+$ is addition, $*$ is multiplication, and $=$ is equality.

Using a first-order language **L**, we can form terms, predicate terms, and formulae. The *terms over* **L** are the following expressions.

(i) Every variable t is a term.

(ii) Every constant symbol $c \in \mathbf{CS}$, is a term.

(iii) If $f \in \mathbf{FS}$ is a function symbol taking n arguments and $\tau_1, \ldots, \tau_n$ are terms already constructed, then $f(\tau_1, \ldots, \tau_n)$ is a term.

The *predicate terms over* **L** are the following expressions.

(i) If $P \in \mathbf{PS}$ is a predicate symbol taking n arguments and $\tau_1, \ldots, \tau_n$ are terms already constructed, then $P(\tau_1, \ldots, \tau_n)$ is a predicate term. When $n = 0$, the predicate symbol P is a predicate term called a propositional symbol.

(ii) When we allow the equality predicate, for any two terms τ_1 and τ_2, the expression $\tau_1 = \tau_2$ is a predicate term. It is usually called an *equation*.

The *(first-order) formulae over* $\mathbf{L}$ are the following expressions.

(i) Every predicate term $P(\tau_1, \ldots, \tau_n)$ is an atomic formula. This includes all propositional letters. We also view $\perp$ (and sometimes $\top$) as an atomic formula.

(ii) When we allow the equality predicate, every equation $\tau_1 = \tau_2$ is an atomic formula.

(iii) If P and Q are formulae already constructed, then $P \Rightarrow Q$, $P \wedge Q$, $P \vee Q$, $\neg P$ are compound formulae. We treat $P \equiv Q$ as an abbreviation for $(P \Rightarrow Q) \wedge (Q \Rightarrow P)$, as before.

(iv) If P is a formula already constructed and t is any variable, then $\forall t P$ and $\exists t P$ are *quantified* compound formulae.

All this can be made very precise but this is quite tedious. Our primary goal is to explain the basic rules of logic and not to teach a full-fledged logic course. We hope that our intuitive explanations will suffice, and we now come to the heart of the matter, the inference rules for the quantifiers. Once again, for a complete treatment, readers are referred to Gallier [19], van Dalen [51], or Huth and Ryan [31].

Unlike the rules for $\Rightarrow, \vee, \wedge$ and $\perp$, which are rather straightforward, the rules for quantifiers are more subtle due to the presence of variables (occurring in terms and predicates). We have to be careful to forbid inferences that would yield "wrong" results and for this *we have to be very precise about the way we use free variables*. More specifically, we have to exercise care when we make *substitutions* of terms for variables in propositions. For example, say we have the predicate "odd," intended to express that a number is odd. Now we can substitute the term $(2y+1)^2$ for x in $\mathrm{odd}(x)$ and obtain

$$\mathrm{odd}((2y+1)^2).$$

More generally, if $P(t_1, t_2, \ldots, t_n)$ is a statement containing the free variables $t_1, \ldots, t_n$ and if $\tau_1, \ldots, \tau_n$ are terms, we can form the new statement

$$P[\tau_1/t_1, \ldots, \tau_n/t_n]$$

obtained by substituting the term τ_i for all free occurrences of the variable t_i, for $i = 1, \ldots, n$. By the way, we denote terms by the Greek letter τ because we use the letter t for a variable and using t for both variables and terms would be confusing.

However, if $P(t_1, t_2, \ldots, t_n)$ contains quantifiers, some bad things can happen; namely, some of the variables occurring in some term τ_i may become quantified when τ_i is substituted for t_i. For example, consider

$$\forall x \exists y\, P(x, y, z)$$

which contains the free variable z and substitute the term $x + y$ for z: we get

$$\forall x \exists y\, P(x, y, x + y).$$

We see that the variables x and y occurring in the term $x+y$ become bound variables after substitution. We say that there is a "capture of variables."

This is not what we intended to happen. To fix this problem, we recall that bound variables are really place holders, so they can be renamed without changing anything. Therefore, we can rename the bound variables x and y in $\forall x \exists y\, P(x,y,z)$ to u and v, getting the statement $\forall u \exists v\, P(u,v,z)$ and now, the result of the substitution is

$$\forall u \exists v\, P(u,v,x+y).$$

Again, all this needs to be explained very carefuly but this can be done.

Finally, here are the inference rules for the quantifiers, first stated in a natural deduction style and then in sequent style. It is assumed that we use two disjoint sets of variables for labeling premises $(x,y,\ldots)$ and free variables $(t,u,v,\ldots)$. As we show, the $\forall$-introduction rule and the $\exists$-elimination rule involve a *crucial restriction* on the occurrences of certain variables. Remember, *variables are terms*.

Definition 2.15. The *inference rules for the quantifiers* are

$\forall$-introduction:

If $\mathcal{D}$ is a deduction tree for $P[u/t]$ from the premises Γ, then

$$\frac{\begin{array}{c}\Gamma\\ \mathcal{D}\\ P[u/t]\end{array}}{\forall t P}$$

is a deduction tree for $\forall t P$ from the premises Γ. Here, u must be a variable that *does not occur free in any of the propositions in* Γ *or in* $\forall t P$. The notation $P[u/t]$ stands for the result of substituting u for all free occurrences of t in P.

Recall that Γ denotes the multiset of premises of the deduction tree $\mathcal{D}$, so if $\mathcal{D}$ only has one node, then $\Gamma = \{P[u/t]\}$ and t should not occur in P.

$\forall$-elimination:

If $\mathcal{D}$ is a deduction tree for $\forall t P$ from the premises Γ, then

$$\frac{\begin{array}{c}\Gamma\\ \mathcal{D}\\ \forall t P\end{array}}{P[\tau/t]}$$

is a deduction tree for $P[\tau/t]$ from the premises Γ. Here τ is an arbitrary term and it is assumed that bound variables in P have been renamed so that none of the variables in τ are captured after substitution.

$\exists$-introduction:

If $\mathcal{D}$ is a deduction tree for $P[\tau/t]$ from the premises Γ, then

$$\begin{array}{c} \Gamma \\ \mathcal{D} \\ P[\tau/t] \\ \hline \exists t P \end{array}$$

is a deduction tree for $\exists t P$ from the premises Γ. As in $\forall$-elimination, τ is an arbitrary term and the same proviso on bound variables in P applies (no capture of variables when τ is substituted).

$\exists$-elimination:

If $\mathcal{D}_1$ is a deduction tree for $\exists t P$ from the premises Γ, and if $\mathcal{D}_2$ is a deduction tree for C from the premises in the multiset Δ and one or more occurrences of $P[u/t]$, then

$$\cfrac{\begin{array}{cc} \Gamma & \Delta, P[u/t]^x \\ \mathcal{D}_1 & \mathcal{D}_2 \\ \exists t P & C \end{array}}{C}\ x$$

is a deduction tree of C from the set of premises in the multiset Γ, Δ. Here, u must be a variable that *does not occur free in any of the propositions in* Δ, $\exists t P$, *or* C, and all premises $P[u/t]$ labeled x are discharged.

In the $\forall$-introduction and the $\exists$-elimination rules, the variable u is called the *eigenvariable* of the inference.

In the above rules, Γ or Δ may be empty; P, C denote arbitrary propositions constructed from a first-order language $\mathbf{L}$; $\mathcal{D}, \mathcal{D}_1, \mathcal{D}_2$ are deductions, possibly a one-node tree; and t is *any* variable.

The system of *first-order classical logic* $\mathcal{N}_c^{\Rightarrow,\vee,\wedge,\perp,\forall,\exists}$ is obtained by adding the above rules to the system of propositional classical logic $\mathcal{N}_c^{\Rightarrow,\vee,\wedge,\perp}$. The system of *first-order intuitionistic logic* $\mathcal{N}_i^{\Rightarrow,\vee,\wedge,\perp,\forall,\exists}$ is obtained by adding the above rules to the system of propositional intuitionistic logic $\mathcal{N}_i^{\Rightarrow,\vee,\wedge,\perp}$. Deduction trees and proof trees are defined as in the propositional case except that the quantifier rules are also allowed.

Using sequents, the quantifier rules in first-order logic are expressed as follows:

Definition 2.16. The *inference rules for the quantifiers in Gentzen-sequent style* are

$$\frac{\Gamma \to P[u/t]}{\Gamma \to \forall t P}\ (\forall\text{-}intro) \qquad \frac{\Gamma \to \forall t P}{\Gamma \to P[\tau/t]}\ (\forall\text{-}elim)$$

where in ($\forall$-*intro*), u does not occur free in Γ or $\forall t P$;

$$\frac{\Gamma \to P[\tau/t]}{\Gamma \to \exists t P}\ (\exists\text{-}intro) \qquad \frac{\Gamma \to \exists t P \quad z\colon P[u/t], \Delta \to C}{\Gamma \cup \Delta \to C}\ (\exists\text{-}elim),$$

where in ($\exists$-*elim*), u does not occur free in Γ, $\exists t P$, or C. Again, t is *any* variable.

The variable u is called the *eigenvariable* of the inference. The systems $\mathcal{NG}_c^{\Rightarrow,\vee,\wedge,\perp,\forall,\exists}$ and $\mathcal{NG}_i^{\Rightarrow,\vee,\wedge,\perp,\forall,\exists}$ are defined from the systems $\mathcal{NG}_c^{\Rightarrow,\vee,\wedge,\perp}$ and $\mathcal{NG}_i^{\Rightarrow,\vee,\wedge,\perp}$, respectively, by adding the above rules. As usual, a *deduction tree* is a either a one-node tree or a tree constructed using the above rules and a *proof tree* is a deduction tree whose conclusion is a sequent with an empty set of premises (a sequent of the form $\emptyset \to P$).

When we say that a proposition P is *provable from* Γ we mean that we can construct a proof tree whose conclusion is P and whose set of premises is Γ in one of the systems $\mathcal{N}_c^{\Rightarrow,\wedge,\vee,\perp,\forall,\exists}$ or $\mathcal{NG}_c^{\Rightarrow,\wedge,\vee,\perp,\forall,\exists}$. Therefore, as in propositional logic, when we use the word "provable" unqualified, we mean provable in *classical logic*. Otherwise, we say *intuitionistically provable*.

It is not hard to show that the proof systems $\mathcal{N}_c^{\Rightarrow,\wedge,\vee,\perp,\forall,\exists}$ and $\mathcal{NG}_c^{\Rightarrow,\wedge,\vee,\perp,\forall,\exists}$ are equivalent (and similarly for $\mathcal{N}_i^{\Rightarrow,\wedge,\vee,\perp,\forall,\exists}$ and $\mathcal{NG}_i^{\Rightarrow,\wedge,\vee,\perp,\forall,\exists}$). We leave the details as Problem 2.16.

A first look at the above rules shows that universal formulae $\forall tP$ behave somewhat like infinite conjunctions and that existential formulae $\exists tP$ behave somewhat like infinite disjunctions.

The $\forall$-introduction rule looks a little strange but the idea behind it is actually very simple: because u is totally unconstrained, if $P[u/t]$ is provable (from Γ), then intuitively $P[u/t]$ holds of any arbitrary object, and so, the statement $\forall tP$ should also be provable (from Γ). Note that the tree

$$\frac{P[u/t]}{\forall tP}$$

is generally *not* a deduction, because the deduction tree above $\forall tP$ is a one-node tree consisting of the single premise $P[u/t]$, and u occurs in $P[u/t]$ unless t does not occur in P.

The meaning of the $\forall$-elimination is that if $\forall tP$ is provable (from Γ), then P holds for all objects and so, in particular for the object denoted by the term τ; that is, $P[\tau/t]$ should be provable (from Γ).

The $\exists$-introduction rule is dual to the $\forall$-elimination rule. If $P[\tau/t]$ is provable (from Γ), this means that the object denoted by τ satisfies P, so $\exists tP$ should be provable (this latter formula asserts the existence of some object satisfying P, and τ is such an object).

The $\exists$-elimination rule is reminiscent of the $\vee$-elimination rule and is a little more tricky. It goes as follows. Suppose that we proved $\exists tP$ (from Γ). Moreover, suppose that for every possible case $P[u/t]$ we were able to prove C (from Γ). Then as we have "exhausted" all possible cases and as we know from the provability of $\exists tP$ that some case must hold, we can conclude that C is provable (from Γ) without using $P[u/t]$ as a premise.

Like the $\vee$-elimination rule, the $\exists$-elimination rule is not very constructive. It allows making a conclusion (C) by considering alternatives *without knowing which one actually occurs*.

Remark: Analogously to disjunction, in (first-order) intuitionistic logic, if an existential statement $\exists tP$ is provable, then from any proof of $\exists tP$, some term τ can be extracted so that $P[\tau/t]$ is provable. Such a term τ is called a *witness*. The witness property is not easy to prove. It follows from the fact that intuitionistic proofs have a normal form (see Section 2.12). However, no such property holds in classical logic.

We can illustrate, again, the fact that classical logic allows for nonconstructive proofs by re-examining the example at the end of Section 2.6. There we proved that if $\sqrt{2}^{\sqrt{2}}$ is rational, then $a = \sqrt{2}$ and $b = \sqrt{2}$ are both irrational numbers such that a^b is rational, and if $\sqrt{2}^{\sqrt{2}}$ is irrational, then $a = \sqrt{2}^{\sqrt{2}}$ and $b = \sqrt{2}$ are both irrational numbers such that a^b is rational. By $\exists$-introduction, we deduce that if $\sqrt{2}^{\sqrt{2}}$ is rational, then there exist some irrational numbers a, b so that a^b is rational, and if $\sqrt{2}^{\sqrt{2}}$ is irrational, then there exist some irrational numbers a, b so that a^b is rational. In classical logic, as $P \vee \neg P$ is provable, by $\vee$-elimination, we just proved that there exist some irrational numbers a and b so that a^b is rational.

However, this argument does not give us explicitly numbers a and b with the required properties. It only tells us that such numbers must exist. Now it turns out that $\sqrt{2}^{\sqrt{2}}$ is indeed irrational (this follows from the Gel'fond–Schneider theorem, a hard theorem in number theory). Furthermore, there are also simpler explicit solutions such as $a = \sqrt{2}$ and $b = \log_2 9$, as the reader should check.

Here is an example of a proof in the system $\mathcal{N}_c^{\Rightarrow,\vee,\wedge,\perp,\forall,\exists}$ (actually, in $\mathcal{N}_i^{\Rightarrow,\vee,\wedge,\perp,\forall,\exists}$) of the formula $\forall t(P \wedge Q) \Rightarrow \forall t P \wedge \forall t Q$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\forall t(P\wedge Q)^x}{P[u/t]\wedge Q[u/t]}}{P[u/t]}}{\forall tP} \qquad \dfrac{\dfrac{\dfrac{\forall t(P\wedge Q)^x}{P[u/t]\wedge Q[u/t]}}{Q[u/t]}}{\forall tQ}}{\forall tP\wedge\forall tQ}}{\forall t(P\wedge Q)\Rightarrow\forall tP\wedge\forall tQ}\; x$$

In the above proof, u is a new variable, that is, a variable that does not occur free in P or Q. We also have used some basic properties of substitutions such as

$$\begin{aligned}
(P\wedge Q)[\tau/t] &= P[\tau/t]\wedge Q[\tau/t]\\
(P\vee Q)[\tau/t] &= P[\tau/t]\vee Q[\tau/t]\\
(P\Rightarrow Q)[\tau/t] &= P[\tau/t]\Rightarrow Q[\tau/t]\\
(\neg P)[\tau/t] &= \neg P[\tau/t]\\
(\forall sP)[\tau/t] &= \forall sP[\tau/t]\\
(\exists sP)[\tau/t] &= \exists sP[\tau/t],
\end{aligned}$$

for any term τ such that no variable in τ is captured during the substitution (in particular, in the last two cases, the variable s does not occur in τ).

The reader should show that $\forall tP \wedge \forall tQ \Rightarrow \forall t(P \wedge Q)$ is also provable in the system

$\mathcal{N}_i^{\Rightarrow,\vee,\wedge,\perp,\forall,\exists}$. However, in general, one can't just replace $\forall$ by $\exists$ (or $\wedge$ by $\vee$) and still obtain provable statements. For example, $\exists tP \wedge \exists tQ \Rightarrow \exists t(P \wedge Q)$ is not provable at all.

Here is an example in which the $\forall$-introduction rule is applied illegally, and thus, yields a statement that is actually false (not provable). In the incorrect "proof" below, P is an atomic predicate symbol taking two arguments (e.g., "parent") and 0 is a constant denoting zero:

$$\dfrac{\dfrac{\dfrac{\dfrac{P(u,0)^x}{\forall tP(t,0)}\ \text{illegal step!}}{P(u,0) \Rightarrow \forall tP(t,0)}\ \text{Implication-Intro } x}{\forall s(P(s,0) \Rightarrow \forall tP(t,0))}\ \text{Forall-Intro}}{P(0,0) \Rightarrow \forall tP(t,0)}\ \text{Forall-Elim}$$

The problem is that the variable u occurs free in the premise $P[u/t,0] = P(u,0)$ and therefore, the application of the $\forall$-introduction rule in the first step is illegal. However, note that this premise is discharged in the second step and so, the application of the $\forall$-introduction rule in the third step is legal. The (false) conclusion of this faulty proof is that $P(0,0) \Rightarrow \forall tP(t,0)$ is provable. Indeed, there are plenty of properties such that the fact that the single instance $P(0,0)$ holds does not imply that $P(t,0)$ holds for all t.

Remark: The above example shows why *it is desirable to have premises that are universally quantified*. A premise of the form $\forall tP$ can be instantiated to $P[u/t]$, using $\forall$-elimination, where u is a brand new variable. Later on, it may be possible to use $\forall$-introduction without running into trouble with free occurrences of u in the premises. But we still have to be very careful when we use $\forall$-introduction or $\exists$-elimination.

Here are some useful equivalences involving quantifiers. The first two are analogous to the de Morgan laws for $\wedge$ and $\vee$.

Proposition 2.13. *The following equivalences are provable in classical first-order logic.*

$$\begin{aligned} \neg\forall tP &\equiv \exists t\neg P \\ \neg\exists tP &\equiv \forall t\neg P \\ \forall t(P \wedge Q) &\equiv \forall tP \wedge \forall tQ \\ \exists t(P \vee Q) &\equiv \exists tP \vee \exists tQ. \end{aligned}$$

In fact, the last three and $\exists t\neg P \Rightarrow \neg\forall tP$ are provable intuitionistically. Moreover, the formulae

$$\exists t(P \wedge Q) \Rightarrow \exists tP \wedge \exists tQ \quad \textit{and} \quad \forall tP \vee \forall tQ \Rightarrow \forall t(P \vee Q)$$

are provable in intuitionistic first-order logic (and thus, also in classical first-order logic).

Proof. Left as an exercise to the reader. $\square$

Before concluding this section, let us give a few more examples of proofs using the rules for the quantifiers. First let us prove that

$$\forall tP \equiv \forall uP[u/t],$$

where u is any variable not free in $\forall tP$ and such that u is not captured during the substitution. This rule allows us to rename bound variables (under very mild conditions). We have the proofs

$$\dfrac{\dfrac{\dfrac{(\forall tP)^{\alpha}}{P[u/t]}}{\forall uP[u/t]}}{\forall tP \Rightarrow \forall uP[u/t]}\ \alpha$$

and

$$\dfrac{\dfrac{\dfrac{(\forall uP[u/t])^{\alpha}}{P[u/t]}}{\forall tP}}{\forall uP[u/t] \Rightarrow \forall tP}\ \alpha$$

Here is now a proof (intuitionistic) of

$$\exists t(P \Rightarrow Q) \Rightarrow (\forall tP \Rightarrow Q),$$

where t does not occur (free or bound) in Q.

$$\dfrac{\dfrac{\dfrac{(\exists t(P \Rightarrow Q))^{z} \qquad \dfrac{(P[u/t] \Rightarrow Q)^{x} \qquad \dfrac{(\forall tP)^{y}}{P[u/t]}}{Q}}{Q}\ x\ (\exists\text{-elim})}{\forall tP \Rightarrow Q}\ y}{\exists t(P \Rightarrow Q) \Rightarrow (\forall tP \Rightarrow Q)}\ z$$

In the above proof, u is a new variable that does not occur in Q, $\forall tP$, or $\exists t(P \Rightarrow Q)$. Because t does not occur in Q, we have

$$(P \Rightarrow Q)[u/t] = P[u/t] \Rightarrow Q.$$

The converse requires (RAA) and is a bit more complicated. Here is a classical proof:

$$
\begin{prooftree}
\AxiomC{$(\neg\exists t(P \Rightarrow Q))^y$}
\AxiomC{$P[u/t]^\alpha, Q^\beta$}
\UnaryInfC{Q}
\RightLabel{α}
\UnaryInfC{$P[u/t] \Rightarrow Q$}
\UnaryInfC{$\exists t(P \Rightarrow Q)$}
\BinaryInfC{$\perp$}
\RightLabel{β}
\UnaryInfC{$\neg Q$}
\AxiomC{$(\forall t P \Rightarrow Q)^x$}
\AxiomC{$(\neg\exists t(P \Rightarrow Q))^y$}
\AxiomC{$\neg P[u/t]^\delta \quad P[u/t]^\gamma$}
\UnaryInfC{$\perp$}
\UnaryInfC{Q}
\RightLabel{γ}
\UnaryInfC{$P[u/t] \Rightarrow Q$}
\UnaryInfC{$\exists t(P \Rightarrow Q)$}
\BinaryInfC{$\perp$}
\RightLabel{δ (RAA)}
\UnaryInfC{$P[u/t]$}
\UnaryInfC{$\forall t P$}
\BinaryInfC{Q}
\BinaryInfC{$\perp$}
\RightLabel{y (RAA)}
\UnaryInfC{$\exists t(P \Rightarrow Q)$}
\RightLabel{x}
\UnaryInfC{$(\forall t P \Rightarrow Q) \Rightarrow \exists t(P \Rightarrow Q)$}
\end{prooftree}
$$

Next, we give intuitionistic proofs of

$$(\exists t P \wedge Q) \Rightarrow \exists t(P \wedge Q)$$

and

$$\exists t(P \wedge Q) \Rightarrow (\exists t P \wedge Q),$$

where t does not occur (free or bound) in Q.

Here is an intuitionistic proof of the first implication:

$$
\begin{prooftree}
\AxiomC{$(\exists t P \wedge Q)^x$}
\UnaryInfC{$\exists t P$}
\AxiomC{$P[u/t]^y$}
\AxiomC{$(\exists t P \wedge Q)^x$}
\UnaryInfC{Q}
\BinaryInfC{$P[u/t] \wedge Q$}
\UnaryInfC{$\exists t(P \wedge Q)$}
\RightLabel{y ($\exists$-elim)}
\BinaryInfC{$\exists t(P \wedge Q)$}
\RightLabel{x}
\UnaryInfC{$(\exists t P \wedge Q) \Rightarrow \exists t(P \wedge Q)$}
\end{prooftree}
$$

In the above proof, u is a new variable that does not occur in $\exists t P$ or Q. Because t does not occur in Q, we have

$$(P \wedge Q)[u/t] = P[u/t] \wedge Q.$$

Here is an intuitionistic proof of the converse:

$$\dfrac{\dfrac{\dfrac{(\exists t(P\wedge Q))^x \qquad \dfrac{\dfrac{(P[u/t]\wedge Q)^y}{P[u/t]}}{\exists tP}}{\exists tP}\ {}_y\,(\exists\text{-elim}) \qquad \dfrac{(\exists t(P\wedge Q))^x \qquad \dfrac{(P[u/t]\wedge Q)^z}{Q}}{Q}\ {}_z\,(\exists\text{-elim})}{\exists tP\wedge Q}}{\exists t(P\wedge Q)\Rightarrow(\exists tP\wedge Q)}\ {}_x$$

Finally, we give a proof (intuitionistic) of

$$(\forall tP\vee Q)\Rightarrow\forall t(P\vee Q),$$

where t does not occur (free or bound) in Q.

$$\dfrac{\dfrac{(\forall tP\vee Q)^z \qquad \dfrac{\dfrac{\dfrac{(\forall tP)^x}{P[u/t]}}{P[u/t]\vee Q}}{\forall t(P\vee Q)} \qquad \dfrac{\dfrac{Q^y}{P[u/t]\vee Q}}{\forall t(P\vee Q)}}{\forall t(P\vee Q)}\ {}_{x,y}\,(\vee\text{-elim})}{(\forall tP\vee Q)\Rightarrow\forall t(P\vee Q)}\ {}_z$$

In the above proof, u is a new variable that does not occur in $\forall tP$ or Q. Because t does not occur in Q, we have

$$(P\vee Q)[u/t]=P[u/t]\vee Q.$$

The converse requires (RAA).

The useful above equivalences (and more) are summarized in the following propositions.

Proposition 2.14. *(1) The following equivalences are provable in classical first-order logic, provided that t does not occur (free or bound) in Q.*

$$\begin{aligned}\forall tP\wedge Q &\equiv \forall t(P\wedge Q)\\ \exists tP\vee Q &\equiv \exists t(P\vee Q)\\ \exists tP\wedge Q &\equiv \exists t(P\wedge Q)\\ \forall tP\vee Q &\equiv \forall t(P\vee Q).\end{aligned}$$

Furthermore, the first three are provable intuitionistically and so is $(\forall tP\vee Q)\Rightarrow\forall t(P\vee Q)$.

(2) The following equivalences are provable in classical logic, provided that t does not occur (free or bound) in P.

$$\begin{aligned}\forall t(P\Rightarrow Q) &\equiv (P\Rightarrow\forall tQ)\\ \exists t(P\Rightarrow Q) &\equiv (P\Rightarrow\exists tQ).\end{aligned}$$

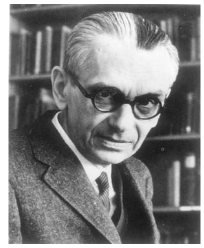
 

Figure 2.12: Andrey N. Kolmogorov, 1903–1987 (left) and Kurt Gödel, 1906–1978 (right)

Furthermore, the first one is provable intuitionistically and so is $\exists t(P \Rightarrow Q) \Rightarrow (P \Rightarrow \exists t Q)$.

(3) The following equivalences are provable in classical logic, provided that t does not occur (free or bound) in Q.

$$\begin{aligned}\forall t(P \Rightarrow Q) &\equiv (\exists t P \Rightarrow Q)\\ \exists t(P \Rightarrow Q) &\equiv (\forall t P \Rightarrow Q).\end{aligned}$$

Furthermore, the first one is provable intuitionistically and so is $\exists t(P \Rightarrow Q) \Rightarrow (\forall t P \Rightarrow Q)$.

Proofs that have not been supplied are left as exercises.

Obviously, every first-order formula that is provable intuitionistically is also provable classically and we know that there are formulae that are provable classically but *not* provable intuitionistically. Therefore, it appears that classical logic is more general than intuitionistic logic. However, this not not quite so because there is a way of translating classical logic into intuitionistic logic. To be more precise, every classical formula A can be translated into a formula A^* where A^* is classically equivalent to A and A is provable classically iff A^* is provable intuitionistically. Various translations are known, all based on a "trick" involving double-negation (This is because $\neg\neg\neg A$ and $\neg A$ are intuitionistically equivalent). Translations were given by Kolmogorov (1925), Gödel (1933), and Gentzen (1933).

For example, Gödel used the following translation.

$$\begin{aligned}A^* &= \neg\neg A, \quad \text{if } A \text{ is atomic},\\ (\neg A)^* &= \neg A^*,\\ (A \wedge B)^* &= (A^* \wedge B^*),\\ (A \Rightarrow B)^* &= \neg(A^* \wedge \neg B^*),\\ (A \vee B)^* &= \neg(\neg A^* \wedge \neg B^*),\\ (\forall x A)^* &= \forall x A^*,\\ (\exists x A)^* &= \neg\forall x \neg A^*.\end{aligned}$$

Actually, if we restrict our attention to propositions (i.e., formulae without quantifiers), a theorem of V. Glivenko (1929) states that if a proposition A is provable classically, then $\neg\neg A$ is provable intuitionistically. In view of these results, the proponents of intuitionistic logic claim that classical logic is really a special case of intuitionistic logic. However, the above translations have some undesirable properties, as noticed by Girard. For more details on all this, see Gallier [14].

2.16 First-Order Theories

The way we presented deduction trees and proof trees may have given our readers the impression that the set of premises Γ was just an auxiliary notion. Indeed, in all of our examples, Γ ends up being empty. However, nonempty Γs are *crucially needed* if we want to develop theories about various kinds of structures and objects, such as the natural numbers, groups, rings, fields, trees, graphs, sets, and the like. Indeed, *we need to make definitions* about the objects we want to study and *we need to state some axioms* asserting the main properties of these objects. *We do this by putting these definitions and axioms in* Γ. Actually, we have to allow Γ to be *infinite* but we still require that our deduction trees be *finite*; they can only use finitely many of the formulae in Γ. We are then interested in all formulae P such that $\Delta \to P$ is provable, where Δ is any finite subset of Γ; the set of all such Ps is called a *theory* (or *first-order theory*). Of course we have the usual problem of consistency: if we are not careful, our theory may be inconsistent, that is, it may consist of all formulae.

Let us give two examples of theories.

Our first example is the *theory of equality*. Indeed, our readers may have noticed that we have avoided dealing with the equality relation. In practice, we can't do that.

Given a language $\mathbf{L}$ with a given supply of constant, function, and predicate symbols, the theory of equality consists of the following formulae taken as axioms.

$$
\begin{aligned}
&\forall x(x = x)\\
&\forall x_1 \cdots \forall x_n \forall y_1 \cdots \forall y_n[(x_1 = y_1 \wedge \cdots \wedge x_n = y_n) \Rightarrow f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n)]\\
&\forall x_1 \cdots \forall x_n \forall y_1 \cdots \forall y_n[(x_1 = y_1 \wedge \cdots \wedge x_n = y_n) \wedge P(x_1, \ldots, x_n) \Rightarrow P(y_1, \ldots, y_n)],
\end{aligned}
$$

for all function symbols (of n arguments) and all predicate symbols (of n arguments), including the equality predicate, $=$, itself.

It is not immediately clear from the above axioms that $=$ is symmetric and transitive but this can be shown easily.

Our second example is the first-order theory of the natural numbers known as *Peano arithmetic* (for short, *PA*).

In this case the language $\mathbf{L}$ consists of the nonlogical symbols $\{0, S, +, *, =\}$. Here, we have the constant 0 (zero), the unary function symbol S (for successor function; the intended meaning is $S(n) = n + 1$) and the binary function symbols $+$ (for addition) and $*$ (for multiplication). In addition to the axioms for the theory of equality we have the

Figure 2.13: Giuseppe Peano, 1858–1932

Figure 2.14: Kurt Gödel with Albert Einstein

following axioms:

$$
\begin{aligned}
&\forall x \neg (S(x) = 0)\\
&\forall x \forall y (S(x) = S(y) \Rightarrow x = y)\\
&\forall x (x + 0 = x)\\
&\forall x \forall y (x + S(y) = S(x + y))\\
&\forall x (x * 0 = 0)\\
&\forall x \forall y (x * S(y) = x * y + x)\\
&[A(0) \land \forall x (A(x) \Rightarrow A(S(x)))] \Rightarrow \forall n A(n),
\end{aligned}
$$

where A is any first-order formula with one free variable.

This last axiom is the *induction axiom*. Observe how $+$ and $*$ are defined recursively in terms of 0 and S and that there are *infinitely many* induction axioms (countably many).

Many properties that hold for the natural numbers (i.e., are true when the symbols $0, S, +, *$ have their usual interpretation and all variables range over the natural numbers) can be proven in this theory (Peano arithmetic), *but not all*. This is another very famous result of Gödel known as *Gödel's first incompleteness theorem* (1931). We give two proofs of Gödel's first incompleteness theorem; one in Section 7.3 using creative and productive sets from recursion theory; the other one using Diophantine definability in Section 8.4.

However, we feel that it should be instructive for the reader to see how simple properties of the natural numbers can be derived (in principle) in Peano arithmetic.

First it is convenient to introduce abbreviations for the terms of the form $S^n(0)$, which represent the natural numbers. Thus, we add a countable supply of constants, $0, 1, 2, 3, \ldots,$ to denote the natural numbers and add the axioms

$$n = S^n(0),$$

for all natural numbers n. We also write $n+1$ for $S(n)$.

Let us illustrate the use of the quantifier rules involving terms ($\forall$-elimination and $\exists$-introduction) by proving some simple properties of the natural numbers, namely, being even or odd. We also prove a property of the natural number that we used before (in the proof that $\sqrt{2}$ is irrational), namely, that *every natural number is either even or odd*. For this, we add the predicate symbols, "even" and "odd", to our language, and assume the following axioms defining these predicates:

$$\begin{aligned}\forall n(\mathrm{even}(n) \equiv \exists k(n = 2 * k))\\ \forall n(\mathrm{odd}(n) \equiv \exists k(n = 2 * k + 1)).\end{aligned}$$

Consider the term, $2 * (m+1) * (m+2) + 1$, where m is any given natural number. We need a few preliminary results.

Proposition 2.15. *The statement* $\mathrm{odd}(2 * (m+1) * (m+2) + 1)$ *is provable in Peano arithmetic.*

As an auxiliary lemma, we first prove

Proposition 2.16. *The formula*

$$\forall x\, \mathrm{odd}(2 * x + 1)$$

is provable in Peano arithmetic.

Proof. Let p be a variable not occurring in any of the axioms of Peano arithmetic (the variable p stands for an arbitrary natural number). From the axiom,

$$\forall n(\mathrm{odd}(n) \equiv \exists k(n = 2 * k + 1)),$$

by $\forall$-elimination where the term $2 * p + 1$ is substituted for the variable n we get

$$\mathrm{odd}(2 * p + 1) \equiv \exists k(2 * p + 1 = 2 * k + 1). \qquad (*)$$

Now we can think of the provable equation $2 * p + 1 = 2 * p + 1$ as

$$(2 * p + 1 = 2 * k + 1)[p/k],$$

so by $\exists$-introduction, we can conclude that

$$\exists k(2 * p + 1 = 2 * k + 1),$$

which, by $(*)$, implies that

$$\mathrm{odd}(2 * p + 1).$$

But now, because p is a variable not occurring free in the axioms of Peano arithmetic, by $\forall$-introduction, we conclude that

$$\forall x\, \mathrm{odd}(2 * x + 1),$$

as claimed. $\square$

Proof of Proposition 2.15. If we use $\forall$-elimination in the above formula where we substitute the term, $\tau = (m + 1) * (m + 2)$, for x, we get

$$\mathrm{odd}(2 * (m + 1) * (m + 2) + 1),$$

as claimed. $\square$

Now we wish to prove

Proposition 2.17. *The formula*

$$\forall n(\mathrm{even}(n) \vee \mathrm{odd}(n))$$

is provable in Peano arithmetic.

Proof. We use the induction principle of Peano arithmetic with

$$A(n) = \mathrm{even}(n) \vee \mathrm{odd}(n).$$

For the base case, $n = 0$, because $0 = 2*0$ (which can be proven from the Peano axioms), we see that $\mathrm{even}(0)$ holds and so $\mathrm{even}(0) \vee \mathrm{odd}(0)$ is proven.

For $n = 1$, because $1 = 2 * 0 + 1$ (which can be proven from the Peano axioms), we see that $\mathrm{odd}(1)$ holds and so $\mathrm{even}(1) \vee \mathrm{odd}(1)$ is proven.

For the induction step, we may assume that $A(n)$ has been proven and we need to prove that $A(n + 1)$ holds.

So, assume that $\mathrm{even}(n) \vee \mathrm{odd}(n)$ holds. We do a proof by cases.

(a) If $\mathrm{even}(n)$ holds, by definition this means that $n = 2k$ for some k and then, $n + 1 = 2k + 1$, which again, by definition means that $\mathrm{odd}(n + 1)$ holds and thus, $\mathrm{even}(n + 1) \vee \mathrm{odd}(n + 1)$ holds.

(b) If $\mathrm{odd}(n)$ holds, by definition this means that $n = 2k + 1$ for some k and then, $n+1 = 2k+2 = 2(k+1)$, which again, by definition means that $\mathrm{even}(n+1)$ holds and thus, $\mathrm{even}(n + 1) \vee \mathrm{odd}(n + 1)$ holds.

By $\vee$-elimination, we conclude that $\mathrm{even}(n + 1) \vee \mathrm{odd}(n + 1)$ holds, establishing the induction step.

Therefore, using induction, we have proven that

$$\forall n(\mathrm{even}(n) \vee \mathrm{odd}(n)),$$

as claimed. $\square$

Actually, we can show that $\mathrm{even}(n)$ and $\mathrm{odd}(n)$ are mutually exclusive as we now prove.

Proposition 2.18. *The formula*

$$\forall n \neg(\mathrm{even}(n) \wedge \mathrm{odd}(n))$$

is provable in Peano arithmetic.

Proof. We prove this by induction. For $n = 0$, the statement $\mathrm{odd}(0)$ means that $0 = 2k + 1 = S(2k)$, for some k. However, the first axiom of Peano arithmetic states that $S(x) \neq 0$ for all x, so we get a contradiction.

For the induction step, assume that $\neg(\mathrm{even}(n) \wedge \mathrm{odd}(n))$ holds. We need to prove that $\neg(\mathrm{even}(n+1) \wedge \mathrm{odd}(n+1))$ holds, and we can do this by using our constructive proof-by-contradiction rule. So, assume that $\mathrm{even}(n+1) \wedge \mathrm{odd}(n+1)$ holds. At this stage, we realize that if we could prove that

$$\forall n(\mathrm{even}(n+1) \Rightarrow \mathrm{odd}(n)) \tag{$*$}$$

and

$$\forall n(\mathrm{odd}(n+1) \Rightarrow \mathrm{even}(n)) \tag{$**$}$$

then $\mathrm{even}(n+1) \wedge \mathrm{odd}(n+1)$ would imply $\mathrm{even}(n) \wedge \mathrm{odd}(n)$, contradicting the assumption $\neg(\mathrm{even}(n) \wedge \mathrm{odd}(n))$. Therefore, the proof is complete if we can prove $(*)$ and $(**)$.

Let's consider the implication $(*)$ leaving the proof of $(**)$ as an exercise.

Assume that $\mathrm{even}(n+1)$ holds. Then $n + 1 = 2k$, for some natural number k. We can't have $k = 0$ because otherwise we would have $n + 1 = 0$, contradicting one of the Peano axioms. But then k is of the form $k = h + 1$ for some natural number h, so

$$n + 1 = 2k = 2(h + 1) = 2h + 2 = (2h + 1) + 1.$$

By the second Peano axiom, we must have

$$n = 2h + 1,$$

which proves that n is odd, as desired.

In that last proof, we made implicit use of the fact that every natural number n different from zero is of the form $n = m + 1$, for some natural number m which is formalized as

$$\forall n((n \neq 0) \Rightarrow \exists m(n = m + 1)).$$

This is easily proven by induction.

Having done all this work, we have finally proven $(*)$ and after proving $(**)$, we will have proven that

$$\forall n \neg(\mathrm{even}(n) \wedge \mathrm{odd}(n)),$$

as claimed. $\square$

It is also easy to prove that

$$\forall n(\text{even}(n) \vee \text{odd}(n))$$

and

$$\forall n \neg(\text{even}(n) \wedge \text{odd}(n))$$

together imply that

$$\forall n(\text{even}(n) \equiv \neg\text{odd}(n)) \quad \text{and} \quad \forall n(\text{odd}(n) \equiv \neg\text{even}(n))$$

are provable, facts that we used several times in Section 2.9. This is because, if

$$\forall x(P \vee Q) \quad \text{and} \quad \forall x \neg(P \wedge Q)$$

can be deduced intuitionistically from a set of premises, Γ, then

$$\forall x(P \equiv \neg Q) \quad \text{and} \quad \forall x(Q \equiv \neg P)$$

can also be deduced intuitionistically from Γ. In this case it also follows that $\forall x(\neg\neg P \equiv P)$ and $\forall x(\neg\neg Q \equiv Q)$ can be deduced intuitionistically from Γ.

Remark: Even though we proved that every nonzero natural number n is of the form $n = m + 1$, for some natural number m, the expression $n - 1$ does not make sense because the predecessor function $n \mapsto n - 1$ has not been defined yet in our logical system. We need to define a function symbol "pred" satisfying the axioms:

$$\begin{aligned} \text{pred}(0) &= 0 \\ \forall n(\text{pred}(n+1) &= n). \end{aligned}$$

For simplicity of notation, we write $n - 1$ instead of $\text{pred}(n)$. Then we can prove that if $k \neq 0$, then $2k - 1 = 2(k-1) + 1$ (which really should be written as $\text{pred}(2k) = 2\text{pred}(k) + 1$). This can indeed be done by induction; we leave the details as an exercise. We can also define substraction, $-$, as a function sastisfying the axioms

$$\begin{aligned} \forall n(n - 0 &= n) \\ \forall n \forall m(n - (m+1) &= \text{pred}(n - m)). \end{aligned}$$

It is then possible to prove the usual properties of subtraction (by induction).

These examples of proofs in the theory of Peano arithmetic illustrate the fact that constructing proofs in an axiomatized theory is a very laborious and tedious process. Many small technical lemmas need to be established from the axioms, which renders these proofs very lengthy and often unintuitive. It is therefore important to build up a database of useful basic facts if we wish to prove, with a certain amount of comfort, properties of objects whose properties are defined by an axiomatic theory (such as the natural numbers). However, when in doubt, we can always go back to the formal theory and try to prove rigorously the facts that we are not sure about, even though this is usually a tedious and painful process. Human provers navigate in a "spectrum of formality," most of the time constructing informal proofs containing quite a few (harmless) shortcuts, sometimes making extra efforts to construct more formalized and rigorous arguments if the need arises.

Now what if the theory of Peano arithmetic were inconsistent! How do know that Peano arithmetic does not imply any contradiction? This is an important and hard question that motivated a lot of the work of Gentzen. An easy answer is that the *standard model* $\mathbb{N}$ of the natural numbers under addition and multiplication validates all the axioms of Peano arithmetic. Therefore, if both P and $\neg P$ could be proven from the Peano axioms, then both P and $\neg P$ would be true in $\mathbb{N}$, which is absurd. To make all this rigorous, we need to define the notion of *truth in a structure*, a notion explained in every logic book. It should be noted that the constructivists will object to the above method for showing the consistency of Peano arithmetic, because it assumes that the infinite set $\mathbb{N}$ exists as a completed entity. Until further notice, we have faith in the consistency of Peano arithmetic (so far, no inconsistency has been found).

Another very interesting theory is *set theory*. There are a number of axiomatizations of set theory and we discuss one of them (ZF) very briefly in Section 2.17.

2.17 Basics Concepts of Set Theory

Having learned some fundamental notions of logic, it is now a good place before proceeding to more interesting things, such as functions and relations, to go through a very quick review of some basic concepts of set theory. This section takes the very "naive" point of view that a set is an unordered collection of objects, without duplicates, the collection being regarded as a single object. Having first-order logic at our disposal, we could formalize set theory very rigorously in terms of axioms. This was done by Zermelo first (1908) and in a more satisfactory form by Zermelo and Fraenkel in 1921, in a theory known as the "Zermelo–Fraenkel" (ZF) axioms. Another axiomatization was given by John von Neumann in 1925 and later improved by Bernays in 1937. A modification of Bernay's axioms was used by Kurt Gödel in 1940. This approach is now known as "von Neumann–Bernays" (VNB) or "Gödel–Bernays" (GB) set theory. There are many books that give an axiomatic presentation of set theory. Among them, we recommend Enderton [11], which we find remarkably clear and elegant, Suppes [48] (a little more advanced), and Halmos [27], a classic (at a more elementary level).

However, it must be said that set theory was first created by Georg Cantor (1845–1918) between 1871 and 1879. However, Cantor's work was not unanimously well received by all mathematicians.

Cantor regarded infinite objects as objects to be treated in much the same way as finite sets, a point of view that was shocking to a number of very prominent mathematicians who bitterly attacked him (among them, the powerful Kronecker). Also, it turns out that some paradoxes in set theory popped up in the early 1900s, in particular, Russell's paradox.

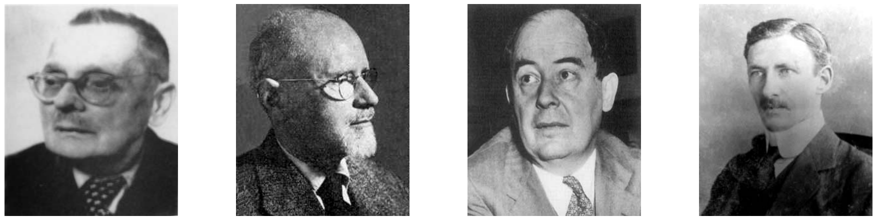


Figure 2.15: Ernst F. Zermelo, 1871–1953 (left), Adolf A. Fraenkel, 1891–1965 (middle left), John von Neumann, 1903–1957 (middle right) and Paul I. Bernays, 1888–1977 (right)

Figure 2.16: Georg F. L. P. Cantor, 1845–1918

Russell's paradox (found by Russell in 1902) has to to with the
"set of all sets that are not members of themselves,"

which we denote by

$$R = \{x \mid x \notin x\}.$$

(In general, the notation $\{x \mid P\}$ stand for the set of all objects satisfying the property P.)

Now, classically, either $R \in R$ or $R \notin R$. However, if $R \in R$, then the definition of R says that $R \notin R$; if $R \notin R$, then again, the definition of R says that $R \in R$.

So, we have a contradiction and the existence of such a set is a paradox. The problem is that we are allowing a property (here, $P(x) = x \notin x$), which is "too wild" and circular in nature. As we show, the way out, as found by Zermelo, is to place a restriction on the

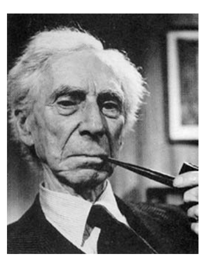


Figure 2.17: Bertrand A. W. Russell, 1872–1970

property P and to also make sure that P picks out elements from some already given set (see the subset axioms below).

The apparition of these paradoxes prompted mathematicians, with Hilbert among its leaders, to put set theory on firmer ground. This was achieved by Zermelo, Fraenkel, von Neumann, Bernays, and Gödel, to name only the major players.

In what follows, we are assuming that we are working in classical logic. The language **L** of set theory consists of the symbols $\{\emptyset, \in, =\}$, where $\emptyset$ is a constant symbol (corresponding to the empty set) and $\in$ is binary predicate symbol (denoting set membership).

In set theory formalized in first-order logic, *every object is a set*. Instead of writing the membership relation as $\in(X, Y)$, we write $X \in Y$, which expresses that the set X belongs to the set Y. To reduce the level of formality, we often denote sets using capital letters and members of sets using lower-case letters, and so we wite $a \in A$ for a belongs to the set A (even though a is also a set). Instead of $\neg(a \in A)$, we write

$$a \notin A.$$

We introduce various operations on sets using definitions involving the logical connectives $\wedge$, $\vee$, $\neg$, $\forall$, and $\exists$.

In order to ensure the existence of some of these sets requires some of the *axioms of set theory*, but we are rather casual about that.

When are two sets A and B equal? This corresponds to the first axiom of set theory, called the

Extensionality Axiom
Two sets A and B are equal iff they have exactly the same elements; that is,

$$\forall x(x \in A \Rightarrow x \in B) \wedge \forall x(x \in B \Rightarrow x \in A).$$

The above says: every element of A is an element of B and conversely.

There is a special set having no elements at all, the *empty set*, denoted $\emptyset$. This is the following.

Empty Set Axiom There is a set having no members. This set is denoted $\emptyset$ and it is characterized by the property

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

Remark: Beginners often wonder whether there is more than one empty set. For example, is the empty set of professors distinct from the empty set of potatoes?

The answer is, by the extensionality axiom, there is only *one* empty set.

Given any two objects a and b, we can form the set $\{a, b\}$ containing exactly these two objects. Amazingly enough, this must also be an axiom:

Pairing Axiom
Given any two objects a and b (think sets), there is a set $\{a, b\}$ having as members just a and b.

Observe that if a and b are identical, then we have the set $\{a, a\}$, which is denoted by $\{a\}$ and is called a *singleton set* (this set has a as its only element).

To form bigger sets, we use the union operation. This too requires an axiom.

Union Axiom (Version 1)
For any two sets A and B, there is a set $A \cup B$ called the *union of A and B* defined by

$$x \in A \cup B \quad \text{iff} \quad (x \in A) \vee (x \in B).$$

This reads, x is a member of $A \cup B$ if either x belongs to A or x belongs to B (or both). We also write

$$A \cup B = \{x \mid x \in A \quad \text{or} \quad x \in B\}.$$

Using the union operation, we can form bigger sets by taking unions with singletons. For example, we can form

$$\{a, b, c\} = \{a, b\} \cup \{c\}.$$

Remark: We can systematically construct bigger and bigger sets by the following method: Given any set A let

$$A^+ = A \cup \{A\}.$$

If we start from the empty set, we obtain sets that can be used to define the natural numbers and the $+$ operation corresponds to the successor function on the natural numbers (i.e., $n \mapsto n + 1$).

Another operation is the power set formation. It is indeed a "powerful" operation, in the sense that it allows us to form very big sets. For this, it is helpful to define the notion of inclusion between sets. Given any two sets, A and B, we say that *A is a subset of B* (or that *A is included in B*), denoted $A \subseteq B$, iff every element of A is also an element of B, that is,

$$\forall x (x \in A \Rightarrow x \in B).$$

We say that *A is a proper subset of B* iff $A \subseteq B$ and $A \neq B$. This implies that that there is some $b \in B$ with $b \notin A$. We usually write $A \subset B$.

Observe that the equality of two sets can be expressed by

$$A = B \quad \text{iff} \quad A \subseteq B \quad \text{and} \quad B \subseteq A.$$

Power Set Axiom
Given any set A, there is a set $\mathcal{P}(A)$ (also denoted 2^A) called the *power set of A* whose members are exactly the subsets of A; that is,

$$X \in \mathcal{P}(A) \quad \text{iff} \quad X \subseteq A.$$

For example, if $A = \{a, b, c\}$, then

$$\mathcal{P}(A) = \{\emptyset, \{a\}, \{b\}, \{c\}, \{a, b\}, \{a, c\}, \{b, c\}, \{a, b, c\}\},$$

a set containing eight elements. Note that the empty set and A itself are always members of $\mathcal{P}(A)$.

Remark: If A has n elements, it is not hard to show that $\mathcal{P}(A)$ has 2^n elements. For this reason, many people, including me, prefer the notation 2^A for the power set of A.

At this stage, we define intersection and complementation. For this, given any set A and given a property P (specified by a first-order formula) we need to be able to define the subset of A consisting of those elements satisfying P. This subset is denoted by

$$\{x \in A \mid P\}.$$

Unfortunately, there are problems with this construction. If the formula P is somehow a circular definition and refers to the subset that we are trying to define, then some paradoxes may arise.

The way out is to place a restriction on the formula used to define our subsets, and this leads to the subset axioms, first formulated by Zermelo. These axioms are also called *comprehension axioms* or *axioms of separation*.

Subset Axioms

For every first-order formula P we have the axiom:

$$\forall A \exists X \forall x (x \in X \quad \text{iff} \quad (x \in A) \wedge P),$$

where P does *not* contain X as a free variable. (However, P may contain x free.)

The subset axioms says that for every set A there is a set X consisting exactly of those elements of A so that P holds. For short, we usually write

$$X = \{x \in A \mid P\}.$$

As an example, consider the formula

$$P(B, x) = x \in B.$$

Then, the subset axiom says

$$\forall A \exists X \forall x (x \in A \wedge x \in B),$$

which means that X is the set of elements that belong both to A and B. This is called the *intersection of A and B*, denoted by $A \cap B$. Note that

$$A \cap B = \{x \mid x \in A \quad \text{and} \quad x \in B\}.$$

We can also define the *relative complement of B in A*, denoted $A - B$, given by the formula $P(B, x) = x \notin B$, so that

$$A - B = \{x \mid x \in A \quad \text{and} \quad x \notin B\}.$$

In particular, if A is any given set and B is any subset of A, the set $A - B$ is also denoted $\overline{B}$ and is called the *complement of* B.

The algebraic properties of union, intersection, and complementation are inherited from the properties of disjunction, conjunction, and negation. The following proposition lists some of the most important properties of union, intersection, and complementation.

Proposition 2.19. *The following equations hold for all sets A, B, C:*

$$\begin{aligned} A \cup \emptyset &= A \\ A \cap \emptyset &= \emptyset \\ A \cup A &= A \\ A \cap A &= A \\ A \cup B &= B \cup A \\ A \cap B &= B \cap A. \end{aligned}$$

The last two assert the commutativity of $\cup$ and $\cap$. We have distributivity of $\cap$ over $\cup$ and of $\cup$ over $\cap$:

$$\begin{aligned} A \cap (B \cup C) &= (A \cap B) \cup (A \cap C) \\ A \cup (B \cap C) &= (A \cup B) \cap (A \cup C). \end{aligned}$$

We have associativity of $\cap$ and $\cup$:

$$\begin{aligned} A \cap (B \cap C) &= (A \cap B) \cap C \\ A \cup (B \cup C) &= (A \cup B) \cup C. \end{aligned}$$

Proof. Use Proposition 2.5. □

Because $\wedge, \vee$, and $\neg$ satisfy the de Morgan laws (remember, we are dealing with classical logic), for any set X, the operations of union, intersection, and complementation on subsets of X satisfy the de Morgan laws.

Proposition 2.20. *For every set X and any two subsets A, B of X, the following identities (de Morgan laws) hold:*

$$\begin{aligned} \overline{\overline{A}} &= A \\ \overline{(A \cap B)} &= \overline{A} \cup \overline{B} \\ \overline{(A \cup B)} &= \overline{A} \cap \overline{B}. \end{aligned}$$

So far, the union axiom only applies to two sets but later on we need to form infinite unions. Thus, it is necessary to generalize our union axiom as follows.

Union Axiom (Final Version)

Given any set X (think of X as a set of sets), there is a set $\bigcup X$ defined so that

$$x \in \bigcup X \quad \text{iff} \quad \exists B(B \in X \wedge x \in B).$$

This says that $\bigcup X$ consists of all elements that belong to some member of X.

If we take $X = \{A, B\}$, where A and B are two sets, we see that

$$\bigcup \{A, B\} = A \cup B,$$

and so, our final version of the union axiom subsumes our previous union axiom which we now discard in favor of the more general version.

Observe that

$$\bigcup \{A\} = A, \quad \bigcup \{A_1, \ldots, A_n\} = A_1 \cup \cdots \cup A_n.$$

and in particular, $\bigcup \emptyset = \emptyset$.

Using the subset axioms, we can also define infinite intersections. For every nonempty set X there is a set $\bigcap X$ defined by

$$x \in \bigcap X \quad \text{iff} \quad \forall B(B \in X \Rightarrow x \in B).$$

The existence of $\bigcap X$ is justified as follows: Because X is nonempty, it contains some set, A; let

$$P(X, x) = \forall B(B \in X \Rightarrow x \in B).$$

Then, the subset axioms asserts the existence of a set Y so that for every x,

$$x \in Y \quad \text{iff} \quad x \in A \quad \text{and} \quad P(X, x)$$

which is equivalent to

$$x \in Y \quad \text{iff} \quad P(X, x).$$

Therefore, the set Y is our desired set, $\bigcap X$.

Observe that

$$\bigcap \{A, B\} = A \cap B, \quad \bigcap \{A_1, \ldots, A_n\} = A_1 \cap \cdots \cap A_n.$$

Note that $\bigcap \emptyset$ is not defined. Intuitively, it would have to be the set of all sets, but such a set does not exist, as we now show. This is basically a version of Russell's paradox.

Theorem 2.21. *(Russell) There is no set of all sets, that is, there is no set to which every other set belongs.*

Proof. Let A be any set. We construct a set B that does not belong to A. If the set of all sets existed, then we could produce a set that does not belong to it, a contradiction. Let

$$B = \{a \in A \mid a \notin a\}.$$

We claim that $B \notin A$. We proceed by contradiction, so assume $B \in A$. However, by the definition of B, we have

$$B \in B \quad \text{iff} \quad B \in A \quad \text{and} \quad B \notin B.$$

Because $B \in A$, the above is equivalent to

$$B \in B \quad \text{iff} \quad B \notin B,$$

which is a contradiction. Therefore, $B \notin A$ and we deduce that there is no set of all sets. $\square$

Remarks:

(1) We should justify why the equivalence $B \in B$ iff $B \notin B$ is a contradiction. What we mean by "a contradiction" is that if the above equivalence holds, then we can derive $\perp$ (falsity) and thus, all propositions become provable. This is because we can show that for any proposition P if $P \equiv \neg P$ is provable, then every proposition is provable. We leave the proof of this fact as an easy exercise for the reader. By the way, this holds classically as well as intuitionistically.

(2) We said that in the subset axioms, the variable X is not allowed to occur free in P. A slight modification of Russell's paradox shows that allowing X to be free in P leads to paradoxical sets. For example, pick A to be any nonempty set and set $P(X, x) = x \notin X$. Then, look at the (alleged) set

$$X = \{x \in A \mid x \notin X\}.$$

As an exercise, the reader should show that X is empty iff X is nonempty,

This is as far as we can go with the elementary notions of set theory that we have introduced so far. In order to proceed further, we need to define relations and functions, which is the object of the next chapter.

The reader may also wonder why we have not yet discussed infinite sets. This is because we don't know how to show that they exist. Again, perhaps surprisingly, this takes another axiom, the *axiom of infinity*. We also have to define when a set is infinite. However, we do not go into this right now. Instead, we accept that the set of natural numbers $\mathbb{N}$ exists and is infinite. Once we have the notion of a function, we will be able to show that other sets are infinite by comparing their "size" with that of $\mathbb{N}$ (This is the purpose of *cardinal numbers*, but this would lead us too far afield).

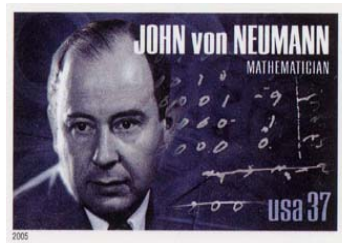


Figure 2.18: John von Neumann

Remark: In an axiomatic presentation of set theory, the natural numbers can be defined from the empty set using the operation $A \mapsto A^+ = A \cup \{A\}$ introduced just after the union axiom. The idea due to von Neumann is that the natural numbers, $0, 1, 2, 3, \ldots$, can be viewed as concise notations for the following sets.

$$
\begin{aligned}
0 &= \emptyset \\
1 &= 0^+ = \{\emptyset\} = \{0\} \\
2 &= 1^+ = \{\emptyset, \{\emptyset\}\} = \{0, 1\} \\
3 &= 2^+ = \{\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}\} = \{0, 1, 2\} \\
&\;\vdots \\
n+1 &= n^+ = \{0, 1, 2, \ldots, n\} \\
&\;\vdots
\end{aligned}
$$

However, the above subsumes induction. Thus, we have to proceed in a different way to avoid circularities.

Definition 2.17. We say that a set X is *inductive* iff

(1) $\emptyset \in X$.

(2) For every $A \in X$, we have $A^+ \in X$.

Axiom of Infinity

There is some inductive set.

Having done this, we make the following.

Definition 2.18. A *natural number* is a set that belongs to every inductive set.

Using the subset axioms, we can show that there is a set whose members are exactly the natural numbers. The argument is very similar to the one used to prove that arbitrary intersections exist. By the axiom of infinity, there is some inductive set, say A. Now consider the property $P(x)$ which asserts that x belongs to every inductive set. By the subset axioms applied to P, there is a set, $\mathbb{N}$, such that

$$x \in \mathbb{N} \quad \text{iff} \quad x \in A \quad \text{and} \quad P(x)$$

and because A is inductive and P says that x belongs to every inductive set, the above is equivalent to

$$x \in \mathbb{N} \quad \text{iff} \quad P(x);$$

that is, $x \in \mathbb{N}$ iff x belongs to every inductive set. Therefore, the set of all natural numbers $\mathbb{N}$ does exist. The set $\mathbb{N}$ is also denoted ω. We can now easily show the following.

Theorem 2.22. *The set $\mathbb{N}$ is inductive and it is a subset of every inductive set.*

Proof. Recall that $\emptyset$ belongs to every inductive set; so, $\emptyset$ is a natural number (0). As $\mathbb{N}$ is the set of natural numbers, $\emptyset$ $(= 0)$ belongs to $\mathbb{N}$. Secondly, if $n \in \mathbb{N}$, this means that n belongs to every inductive set (n is a natural number), which implies that $n^+ = n + 1$ belongs to every inductive set, which means that $n+1$ is a natural number, that is, $n+1 \in \mathbb{N}$. Because $\mathbb{N}$ is the set of natural numbers and because every natural number belongs to every inductive set, we conclude that $\mathbb{N}$ is a subset of every inductive set. $\square$

It would be tempting to view $\mathbb{N}$ as the intersection of the family of inductive sets, but unfortunately this family is not a set; it is too "big" to be a set.

As a consequence of the above fact, we obtain the following.

Induction Principle for $\mathbb{N}$: Any inductive subset of $\mathbb{N}$ is equal to $\mathbb{N}$ itself.

Now, in our setting, $0 = \emptyset$ and $n^+ = n + 1$, so the above principle can be restated as follows.

Induction Principle for $\mathbb{N}$ **(Version 2)**: For any subset, $S \subseteq \mathbb{N}$, if $0 \in S$ and $n + 1 \in S$ whenever $n \in S$, then $S = \mathbb{N}$.

We show how to rephrase this induction principle a little more conveniently in terms of the notion of function in the next chapter.

Remarks:

1. We still don't know what an infinite set is or, for that matter, that $\mathbb{N}$ is infinite.

2. Zermelo–Fraenkel set theory (+ Choice) has three more axioms that we did not discuss: The *axiom of choice*, the *replacement axioms* and the *regularity axiom*. For our purposes, only the axiom of choice is needed. Let us just say that the replacement axioms are needed to deal with ordinals and cardinals and that the regularity axiom is needed to show that every set is grounded. For more about these axioms, see Enderton [11], Chapter 7. The regularity axiom also implies that no set can be a member of itself, an eventuality that is not ruled out by our current set of axioms.

As we said at the beginning of this section, set theory can be axiomatized in first-order logic. To illustrate the generality and expressiveness of first-order logic, we conclude this section by stating the axioms of *Zermelo–Fraenkel set theory* (for short, *ZF*) as first-order formulae. The language of Zermelo–Fraenkel set theory consists of the constant $\emptyset$ (for the empty set), the equality symbol, and of the binary predicate symbol $\in$ for set membership. It is convenient to abbreviate $\neg(x = y)$ as $x \neq y$ and $\neg(x \in y)$ as $x \notin y$. The axioms are the equality axioms plus the following seven axioms.

$$\begin{aligned}
&\forall A \forall B (\forall x (x \in A \equiv x \in B) \Rightarrow A = B)\\
&\forall x (x \notin \emptyset)\\
&\forall a \forall b \exists Z \forall x (x \in Z \equiv (x = a \vee x = b))\\
&\forall X \exists Y \forall x (x \in Y \equiv \exists B (B \in X \wedge x \in B))\\
&\forall A \exists Y \forall X (X \in Y \equiv \forall z (z \in X \Rightarrow z \in A))\\
&\forall A \exists X \forall x (x \in X \equiv (x \in A) \wedge P)\\
&\exists X (\emptyset \in X \wedge \forall y (y \in X \Rightarrow y \cup \{y\} \in X)),
\end{aligned}$$

where P is any first-order formula that does not contain X free.

- Axiom (1) is the extensionality axiom.
- Axiom (2) is the empty set axiom.
- Axiom (3) asserts the existence of a set Y whose only members are a and b. By extensionality, this set is unique and it is denoted $\{a, b\}$. We also denote $\{a, a\}$ by $\{a\}$.
- Axiom (4) asserts the existence of set Y which is the union of all the sets that belong to X. By extensionality, this set is unique and it is denoted $\bigcup X$. When $X = \{A, B\}$, we write $\bigcup\{A, B\} = A \cup B$.
- Axiom (5) asserts the existence of set Y which is the set of all subsets of A (the power set of A). By extensionality, this set is unique and it is denoted $\mathcal{P}(A)$ or 2^A.
- Axioms (6) are the subset axioms (or axioms of separation).
- Axiom (7) is the infinity axiom, stated using the abbreviations introduced above.

For a comprehensive treatment of axiomatic theory (including the missing three axioms), see Enderton [11] and Suppes [48].

2.18 Summary

The main goal of this chapter is to describe precisely the logical rules used in mathematical reasoning and the notion of a mathematical proof. A brief introduction to set theory is also provided. We decided to describe the rules of reasoning in a formalism known as a natural deduction system because the logical rules of such a system mimic rather closely the informal rules that (nearly) everybody uses when constructing a proof in everyday life. Another advantage of natural deduction systems is that it is very easy to present various versions of the rules involving negation and thus, to explain why the "proof-by-contradiction" proof rule or the "law of the excluded middle" allow for the derivation of "nonconstructive" proofs. This is a subtle point often not even touched in traditional presentations of logic. However, inasmuch as most of our readers write computer programs and expect that their programs will not just promise to give an answer but will actually produce results, we feel that they will grasp rather easily the difference between constructive and nonconstructive proofs, and appreciate the latter, even if they are harder to find.

- We describe the syntax of *propositional logic*.
- The proof rules for *implication* are defined in a *natural deduction system* (Prawitz-style).
- *Deductions* proceed from *assumptions* (or *premises*) using *inference rules*.
- The process of *discharging* (or *closing*) a premise is explained. A *proof* is a deduction in which all the premises have been discharged.
- We explain how we can *search* for a proof using a combined bottom-up and top-down process.
- We propose another mechanism for decribing the process of discharging a premise and this leads to a formulation of the rules in terms of *sequents* and to a *Gentzen system*.
- We introduce falsity $\perp$ and negation $\neg P$ as an abbrevation for $P \Rightarrow \perp$. We describe the inference rules for conjunction, disjunction, and negation, in both Prawitz style and Gentzen-sequent style *natural deduction systems*
- One of the rules for negation is the *proof-by-contradiction* rule (also known as *RAA*).
- We define *intuitionistic* and *classical* logic.
- We introduce the notion of a *constructive* (or *intuitionistic*) proof and discuss the two nonconstructive culprits: $P \vee \neg P$ (the *law of the excluded middle*) and $\neg\neg P \Rightarrow P$ (*double-negation rule*).
- We show that $P \vee \neg P$ and $\neg\neg P \Rightarrow P$ are provable in classical logic
- We clear up some potential confusion involving the various versions of the rules regarding negation.

    1. RAA is not a special case of $\neg$-introduction.

2. RAA is not equivalent to $\perp$-elimination; in fact, it implies it.
3. Not all propositions of the form $P \vee \neg P$ are provable in intuitionistic logic. However, RAA holds in intuitionistic logic plus all propositions of the form $P \vee \neg P$.
4. We define *double-negation elimination*.

- We present the *de Morgan laws* and prove their validity in classical logic.
- We present the *proof-by-contrapositive rule* and show that it is valid in classical logic.
- We give some examples of proofs of "real" statements.
- We give an example of a nonconstructive proof of the statement: there are two irrational numbers, a and b, so that a^b is rational.
- We explain the *truth-value semantics* of propositional logic.
- We define the *truth tables* for the propositional connectives
- We define the notions of *satisfiability*, *unsatisfiability*, *validity*, and *tautology*.
- We define the *satisfiability problem* and the *validity problem* (for classical propositional logic).
- We mention the *NP-completeness* of satisfiability.
- We discuss *soundness* (or *consistency*) and *completeness*.
- We state the *soundness and completeness theorems* for propositional classical logic formulated in natural deduction.
- We explain how to use *counterexamples* to prove that certain propositions are not provable.
- We give a brief introduction to *Kripke semantics* for propositional intuitionistic logic.
- We define *Kripke models* (based on a *set of worlds*).
- We define *validity* in a Kripke model.
- We state the the *soundness and completeness theorems* for propositional intuitionistic logic formulated in natural deduction.
- We add *first-order quantifiers* ("for all" $\forall$ and "there exists" $\exists$) to the language of propositional logic and define *first-order logic*.
- We describe *free* and *bound* variables.

- We give inference rules for the quantifiers in Prawitz-style and Gentzen sequent-style *natural deduction systems*.
- We explain the *eigenvariable restriction* in the $\forall$-introduction and $\exists$-elimination rules.
- We prove some "de Morgan"-type rules for the quantified formulae valid in classical logic.
- We discuss the nonconstructiveness of proofs of certain existential statements.
- We explain briefly how classical logic can be translated into intuitionistic logic (the Gödel translation).
- We define *first-order theories* and give the example of *Peano arithmetic*.
- We revisit the *decision problem* and mention the *undecidability of the decision problem* for first-order logic (*Church's theorem*).
- We discuss the notion of *detours* in proofs and the notion of *proof normalization*.
- We mention *strong normalization*.
- We mention the correspondence between propositions and types and proofs and typed λ-terms (the *Curry–Howard isomorphism*).
- We mention *Gödel's completeness theorem* for first-order logic.
- Again, we mention the use of *counterexamples*.
- We mention *Gödel's incompleteness theorem*.
- We present informally the axioms of *Zermelo–Fraenkel set theory* (ZF).
- We present *Russell's paradox*, a warning against "self-referential" definitions of sets.
- We define the *empty set* ($\emptyset$), the set $\{a, b\}$, whose elements are a and b, the *union* $A \cup B$, of two sets A and B, and the *power set* 2^A, of A.
- We state carefully Zermelo's *subset axioms* for defining the subset $\{x \in A \mid P\}$ of elements of a given set A satisfying a property P.
- Then, we define the *intersection* $A \cap B$, and the *relative complement* $A - B$, of two sets A and B.
- We also define the *union* $\bigcup A$ and the *intersection* $\bigcap A$, of a set of sets A.
- We show that one should avoid sets that are "too big;" in particular, we prove that there is no *set of all sets*.

- We define the *natural numbers* “a la Von Neumann.”
- We define *inductive sets* and state the *axiom of infinity*.
- We show that the natural numbers form an inductive set $\mathbb{N}$, and thus, obtain an *induction principle for* $\mathbb{N}$.
- We summarize the axioms of Zermelo–Fraenkel set theory in first-order logic.

Problems

Problem 2.1. (a) Give a proof of the proposition $P \Rightarrow (Q \Rightarrow P)$ in the system $\mathcal{N}_m^{\Rightarrow}$.

(b) Prove that if there are deduction trees of $P \Rightarrow Q$ and $Q \Rightarrow R$ from the set of premises Γ in the system $\mathcal{N}_m^{\Rightarrow}$, then there is a deduction tree for $P \Rightarrow R$ from Γ in $\mathcal{N}_m^{\Rightarrow}$.

Problem 2.2. Give a proof of the proposition $(P \Rightarrow Q) \Rightarrow ((P \Rightarrow (Q \Rightarrow R)) \Rightarrow (P \Rightarrow R))$ in the system $\mathcal{N}_m^{\Rightarrow}$.

Problem 2.3. (a) Prove the “de Morgan” laws in classical logic:

$$\neg(P \wedge Q) \equiv \neg P \vee \neg Q$$
$$\neg(P \vee Q) \equiv \neg P \wedge \neg Q.$$

(b) Prove that $\neg(P \vee Q) \equiv \neg P \wedge \neg Q$ is also provable in intuitionistic logic.

(c) Prove that the proposition $(P \wedge \neg Q) \Rightarrow \neg(P \Rightarrow Q)$ is provable in intuitionistic logic and $\neg(P \Rightarrow Q) \Rightarrow (P \wedge \neg Q)$ is provable in classical logic.

Problem 2.4. (a) Show that $P \Rightarrow \neg\neg P$ is provable in intuitionistic logic.

(b) Show that $\neg\neg\neg P$ and $\neg P$ are equivalent in intuitionistic logic.

Problem 2.5. Recall that an integer is *even* if it is divisible by 2, that is, if it can be written as $2k$, where $k \in \mathbb{Z}$. An integer is *odd* if it is not divisible by 2, that is, if it can be written as $2k + 1$, where $k \in \mathbb{Z}$. Prove the following facts.

(a) The sum of even integers is even.

(b) The sum of an even integer and of an odd integer is odd.

(c) The sum of two odd integers is even.

(d) The product of odd integers is odd.

(e) The product of an even integer with any integer is even.

Problem 2.6. (a) Show that if we assume that all propositions of the form

$$P \Rightarrow (Q \Rightarrow R)$$

are axioms (where P, Q, R are arbitrary propositions), then *every proposition* is provable.

(b) Show that if P is provable (intuitionistically or classically), then $Q \Rightarrow P$ is also provable for *every* proposition Q.

Problem 2.7. (a) Give intuitionistic proofs for the equivalences

$$\begin{aligned}
&P \vee P \equiv P \\
&P \wedge P \equiv P \\
&P \vee Q \equiv Q \vee P \\
&P \wedge Q \equiv Q \wedge P.
\end{aligned}$$

(b) Give intuitionistic proofs for the equivalences

$$\begin{aligned}
&P \wedge (P \vee Q) \equiv P \\
&P \vee (P \wedge Q) \equiv P.
\end{aligned}$$

Problem 2.8. Give intuitionistic proofs for the propositions

$$\begin{aligned}
&P \Rightarrow (Q \Rightarrow (P \wedge Q)) \\
&(P \Rightarrow Q) \Rightarrow ((P \Rightarrow \neg Q) \Rightarrow \neg P) \\
&(P \Rightarrow R) \Rightarrow ((Q \Rightarrow R) \Rightarrow ((P \vee Q) \Rightarrow R)).
\end{aligned}$$

Problem 2.9. Prove that the following equivalences are provable intuitionistically:

$$\begin{aligned}
P \wedge (P \Rightarrow Q) \;&\equiv\; P \wedge Q \\
Q \wedge (P \Rightarrow Q) \;&\equiv\; Q \\
\big(P \Rightarrow (Q \wedge R)\big) \;&\equiv\; \big((P \Rightarrow Q) \wedge (P \Rightarrow R)\big).
\end{aligned}$$

Problem 2.10. Give intuitionistic proofs for

$$\begin{aligned}
&(P \Rightarrow Q) \Rightarrow \neg\neg(\neg P \vee Q) \\
&\neg\neg(\neg\neg P \Rightarrow P).
\end{aligned}$$

Problem 2.11. Give an intuitionistic proof for $\neg\neg(P \vee \neg P)$.

Problem 2.12. Give intuitionistic proofs for the propositions

$$(P \vee \neg P) \Rightarrow (\neg\neg P \Rightarrow P) \quad \text{and} \quad (\neg\neg P \Rightarrow P) \Rightarrow (P \vee \neg P).$$

Hint. For the second implication, you may want to use Problem 2.11.

Problem 2.13. Give intuitionistic proofs for the propositions

$$(P \Rightarrow Q) \Rightarrow \neg\neg(\neg P \lor Q) \quad \text{and} \quad (\neg P \Rightarrow Q) \Rightarrow \neg\neg(P \lor Q).$$

Problem 2.14. (1) Design an algorithm for converting a deduction of a proposition P in the system $\mathcal{N}_i^{\Rightarrow,\wedge,\vee,\perp}$ into a deduction in the system $\mathcal{NG}_i^{\Rightarrow,\wedge,\vee,\perp}$.

(2) Design an algorithm for converting a deduction of a proposition P in the system $\mathcal{N}_c^{\Rightarrow,\wedge,\vee,\perp}$ into a deduction in the system $\mathcal{NG}_c^{\Rightarrow,\wedge,\vee,\perp}$.

(3) Design an algorithm for converting a deduction of a proposition P in the system $\mathcal{NG}_i^{\Rightarrow,\wedge,\vee,\perp}$ into a deduction in the system $\mathcal{N}_i^{\Rightarrow,\wedge,\vee,\perp}$.

(4) Design an algorithm for converting a deduction of a proposition P in the system $\mathcal{NG}_c^{\Rightarrow,\wedge,\vee,\perp}$ into a deduction in the system $\mathcal{N}_c^{\Rightarrow,\wedge,\vee,\perp}$.

Hint. Use induction on deduction trees.

Problem 2.15. Prove that the following version of the $\vee$-elimination rule formulated in Gentzen-sequent style is a consequence of the rules of intuitionistic logic:

$$\frac{\Gamma, x\colon P \to R \quad \Gamma, y\colon Q \to R}{\Gamma, z\colon P \lor Q \to R}$$

Conversely, if we assume that the above rule holds, then prove that the $\vee$-elimination rule

$$\frac{\Gamma \to P \lor Q \quad \Gamma, x\colon P \to R \quad \Gamma, y\colon Q \to R}{\Gamma \to R} \quad (\vee\text{-}elim)$$

follows from the rules of intuitionistic logic (of course, excluding the $\vee$-elimination rule).

Problem 2.16. (1) Give algorithms for converting a deduction in $\mathcal{N}_c^{\Rightarrow,\wedge,\vee,\perp,\forall,\exists}$ to a deduction in $\mathcal{NG}_c^{\Rightarrow,\wedge,\vee,\perp,\forall,\exists}$ and vice-versa.

(2) Give algorithms for converting a deduction in $\mathcal{N}_i^{\Rightarrow,\wedge,\vee,\perp,\forall,\exists}$ to a deduction in $\mathcal{NG}_i^{\Rightarrow,\wedge,\vee,\perp,\forall,\exists}$ and vice-versa.

Problem 2.17. (a) Give intuitionistic proofs for the distributivity of $\wedge$ over $\vee$ and of $\vee$ over $\wedge$:

$$\begin{aligned} P \wedge (Q \vee R) &\equiv (P \wedge Q) \vee (P \wedge R) \\ P \vee (Q \wedge R) &\equiv (P \vee Q) \wedge (P \vee R). \end{aligned}$$

(b) Give intuitionistic proofs for the associativity of $\wedge$ and $\vee$:

$$\begin{aligned} P \wedge (Q \wedge R) &\equiv (P \wedge Q) \wedge R \\ P \vee (Q \vee R) &\equiv (P \vee Q) \vee R. \end{aligned}$$

Problem 2.18. Recall that in Problem 2.1 we proved that if $P \Rightarrow Q$ and $Q \Rightarrow R$ are provable, then $P \Rightarrow R$ is provable. Deduce from this fact that if $P \equiv Q$ and $Q \equiv R$ hold, then $P \equiv R$ holds (intuitionistically or classically).

Prove that if $P \equiv Q$ holds then $Q \equiv P$ holds (intuitionistically or classically). Finally, check that $P \equiv P$ holds (intuitionistically or classically).

Problem 2.19. Prove (intuitionistically or classically) that if $P_1 \Rightarrow Q_1$ and $P_2 \Rightarrow Q_2$ then

1. $(P_1 \wedge P_2) \Rightarrow (Q_1 \wedge Q_2)$
2. $(P_1 \vee P_2) \Rightarrow (Q_1 \vee Q_2)$.

(b) Prove (intuitionistically or classically) that if $Q_1 \Rightarrow P_1$ and $P_2 \Rightarrow Q_2$ then

1. $(P_1 \Rightarrow P_2) \Rightarrow (Q_1 \Rightarrow Q_2)$
2. $\neg P_1 \Rightarrow \neg Q_1$.

(c) Prove (intuitionistically or classically) that if $P \Rightarrow Q$, then

1. $\forall t P \Rightarrow \forall t Q$
2. $\exists t P \Rightarrow \exists t Q$.

(d) Prove (intuitionistically or classically) that if $P_1 \equiv Q_1$ and $P_2 \equiv Q_2$ then

1. $(P_1 \wedge P_2) \equiv (Q_1 \wedge Q_2)$
2. $(P_1 \vee P_2) \equiv (Q_1 \vee Q_2)$
3. $(P_1 \Rightarrow P_2) \equiv (Q_1 \Rightarrow Q_2)$
4. $\neg P_1 \equiv \neg Q_1$
5. $\forall t P_1 \equiv \forall t Q_1$
6. $\exists t P_1 \equiv \exists t Q_1$.

Problem 2.20. Show that the following are provable in classical first-order logic:

$$\begin{aligned} \neg \forall t P &\equiv \exists t \neg P \\ \neg \exists t P &\equiv \forall t \neg P \\ \forall t (P \wedge Q) &\equiv \forall t P \wedge \forall t Q \\ \exists t (P \vee Q) &\equiv \exists t P \vee \exists t Q. \end{aligned}$$

(b) Moreover, show that the propositions $\exists t (P \wedge Q) \Rightarrow \exists t P \wedge \exists t Q$ and $\forall t P \vee \forall t Q \Rightarrow \forall t (P \vee Q)$ are provable in intuitionistic first-order logic (and thus, also in classical first-order logic).

(c) Prove intuitionistically that

$$\exists x \forall y P \Rightarrow \forall y \exists x P.$$

Give an informal argument to the effect that the converse, $\forall y \exists x P \Rightarrow \exists x \forall y P$, is not provable, even classically.

Problem 2.21. (a) Assume that Q is a formula that does **not** contain the variable t (free or bound). Give a classical proof of

$$\forall t(P \vee Q) \Rightarrow (\forall t P \vee Q).$$

(b) If P is a proposition, write $P(x)$ for $P[x/t]$ and $P(y)$ for $P[y/t]$, where x and y are distinct variables that do not occur in the orginal proposition P. Give an intuitionistic proof for

$$\neg \forall x \exists y(\neg P(x) \wedge P(y)).$$

(c) Give a classical proof for

$$\exists x \forall y(P(x) \vee \neg P(y)).$$

Hint. Negate the above, then use some identities we've shown (such as de Morgan) and reduce the problem to part (b).

Problem 2.22. (a) Let $X = \{X_i \mid 1 \leq i \leq n\}$ be a finite family of sets. Prove that if $X_{i+1} \subseteq X_i$ for all i, with $1 \leq i \leq n-1$, then

$$\bigcap X = X_n.$$

Prove that if $X_i \subseteq X_{i+1}$ for all i, with $1 \leq i \leq n-1$, then

$$\bigcup X = X_n.$$

(b) Recall that $\mathbb{N}_+ = \mathbb{N} - \{0\} = \{1, 2, 3, \ldots, n, \ldots\}$. Give an example of an infinite family of sets, $X = \{X_i \mid i \in \mathbb{N}_+\}$, such that

1. $X_{i+1} \subseteq X_i$ for all $i \geq 1$.

2. X_i is infinite for every $i \geq 1$.

3. $\bigcap X$ has a single element.

(c) Give an example of an infinite family of sets, $X = \{X_i \mid i \in \mathbb{N}_+\}$, such that

1. $X_{i+1} \subseteq X_i$ for all $i \geq 1$.

2. X_i is infinite for every $i \geq 1$.

3. $\bigcap X = \emptyset$.

Problem 2.23. Prove that the following propositions are provable intuitionistically:

$$(P \Rightarrow \neg P) \equiv \neg P, \qquad (\neg P \Rightarrow P) \equiv \neg\neg P.$$

Use these to conlude that if the equivalence $P \equiv \neg P$ is provable intuitionistically, then *every* proposition is provable (intuitionistically).

Problem 2.24. (1) Prove that if we assume that all propositions of the form,

$$((P \Rightarrow Q) \Rightarrow P) \Rightarrow P,$$

are axioms (Peirce's law), then $\neg\neg P \Rightarrow P$ becomes provable in intuitionistic logic. Thus, another way to get classical logic from intuitionistic logic is to add Peirce's law to intuitionistic logic.
Hint. Pick Q in a suitable way and use Problem 2.23.

(2) Prove $((P \Rightarrow Q) \Rightarrow P) \Rightarrow P$ in classical logic.
Hint. Use the de Morgan laws.

Problem 2.25. Let A be any nonempty set. Prove that the definition

$$X = \{a \in A \mid a \notin X\}$$

yields a "set," X, such that X is empty iff X is nonempty and therefore does not define a set, after all.

Problem 2.26. Prove the following fact: if

$$\begin{array}{c} \Gamma \\ \mathcal{D}_1 \\ P \vee Q \end{array} \quad \text{and} \quad \begin{array}{c} \Gamma, R \\ \mathcal{D}_2 \\ Q \end{array}$$

are deduction trees provable intuitionistically, then there is a deduction tree

$$\begin{array}{c} \Gamma, P \Rightarrow R \\ \mathcal{D} \\ Q \end{array}$$

for Q from the premises in $\Gamma \cup \{P \Rightarrow S\}$.

Problem 2.27. Recall that the constant $\top$ stands for **true**. So, we add to our proof systems (intuitionistic and classical) all axioms of the form

$$\frac{\overbrace{P_1, \ldots, P_1}^{k_1}, \ldots, \overbrace{P_i, \ldots, P_i}^{k_i}, \ldots, \overbrace{P_n, \ldots, P_n}^{k_n}}{\top}$$

where $k_i \geq 1$ and $n \geq 0$; note that $n = 0$ is allowed, which amounts to the one-node tree, $\top$.

(a) Prove that the following equivalences hold intuitionistically.

$$\begin{aligned} P \lor \top &\equiv \top \\ P \land \top &\equiv P. \end{aligned}$$

Prove that if P is intuitionistically (or classically) provable, then $P \equiv \top$ is also provable intuitionistically (or classically). In particular, in classical logic,
$P \lor \neg P \equiv \top$. Also prove that

$$\begin{aligned} P \lor \perp &\equiv P \\ P \land \perp &\equiv \perp \end{aligned}$$

hold intuitionistically.

(b) In the rest of this problem, we are dealing only with classical logic. The connective *exclusive or*, denoted $\oplus$, is defined by

$$P \oplus Q \equiv (P \land \neg Q) \lor (\neg P \land Q).$$

In solving the following questions, you will find that constructing proofs using the rules of classical logic is very tedious because these proofs are very long. Instead, use some identities from previous problems.

Prove the equivalence

$$\neg P \equiv P \oplus \top.$$

(c) Prove that

$$\begin{aligned} P \oplus P &\equiv \perp \\ P \oplus Q &\equiv Q \oplus P \\ (P \oplus Q) \oplus R &\equiv P \oplus (Q \oplus R). \end{aligned}$$

(d) Prove the equivalence

$$P \lor Q \equiv (P \land Q) \oplus (P \oplus Q).$$

Problem 2.28. Give a classical proof of

$$\neg(P \Rightarrow \neg Q) \Rightarrow (P \land Q).$$

Problem 2.29. (a) Prove that the rule

$$\frac{\begin{array}{cc} \Gamma & \Delta \\ \mathcal{D}_1 & \mathcal{D}_2 \\ P \Rightarrow Q & \neg Q \end{array}}{\neg P}$$

can be derived from the other rules of intuitionistic logic.

(b) Give an intuitionistic proof of $\neg P$ from $\Gamma = \{\neg(\neg P \vee Q), P \Rightarrow Q\}$ or equivalently, an intuitionistic proof of

$$\Big(\neg(\neg P \vee Q) \wedge (P \Rightarrow Q)\Big) \Rightarrow \neg P.$$

Problem 2.30. (a) Give intuitionistic proofs for the equivalences

$$\exists x \exists y P \equiv \exists y \exists x P \quad \text{and} \quad \forall x \forall y P \equiv \forall y \forall x P.$$

(b) Give intuitionistic proofs for

$$(\forall t P \wedge Q) \Rightarrow \forall t(P \wedge Q) \quad \text{and} \quad \forall t(P \wedge Q) \Rightarrow (\forall t P \wedge Q),$$

where t does not occur (free or bound) in Q.

(c) Give intuitionistic proofs for

$$(\exists t P \vee Q) \Rightarrow \exists t(P \vee Q) \quad \text{and} \quad \exists t(P \vee Q) \Rightarrow (\exists t P \vee Q),$$

where t does not occur (free or bound) in Q.

Problem 2.31. An integer, $n \in \mathbb{Z}$, is divisible by 3 iff $n = 3k$, for some $k \in \mathbb{Z}$. Thus (by the division theorem), an integer, $n \in \mathbb{Z}$, is not divisible by 3 iff it is of the form $n = 3k+1, 3k+2$, for some $k \in \mathbb{Z}$ (you don't have to prove this).

Prove that for any integer, $n \in \mathbb{Z}$, if n^2 is divisible by 3, then n is divisible by 3.

Hint. Prove the contrapositive. If n of the form $n = 3k+1, 3k+2$, then so is n^2 (for a different k).

Problem 2.32. Use Problem 2.31 to prove that $\sqrt{3}$ is irrational, that is, $\sqrt{3}$ can't be written as $\sqrt{3} = p/q$, with $p, q \in \mathbb{Z}$ and $q \neq 0$.

Problem 2.33. Give an intuitionistic proof of the proposition

$$\big((P \Rightarrow R) \wedge (Q \Rightarrow R)\big) \equiv \big((P \vee Q) \Rightarrow R\big).$$

Problem 2.34. Give an intuitionistic proof of the proposition

$$\big((P \wedge Q) \Rightarrow R\big) \equiv \big(P \Rightarrow (Q \Rightarrow R)\big).$$

Problem 2.35. (a) Give an intuitionistic proof of the proposition
$(P \wedge Q) \Rightarrow (P \vee Q)$.

(b) Prove that the proposition $(P \vee Q) \Rightarrow (P \wedge Q)$ is not valid, where P, Q, are propositional symbols.

(c) Prove that the proposition $(P \vee Q) \Rightarrow (P \wedge Q)$ is not provable in general and that if we assume that *all* propositions of the form $(P \vee Q) \Rightarrow (P \wedge Q)$ are axioms, then *every* proposition becomes provable intuitionistically.

Problem 2.36. Give the details of the proof of Proposition 2.6; namely, if a proposition P is provable in the system $\mathcal{N}_c^{\Rightarrow,\wedge,\vee,\perp}$ (or $\mathcal{NG}_c^{\Rightarrow,\wedge,\vee,\perp}$), then it is valid (according to the truth value semantics).

Problem 2.37. Give the details of the proof of Theorem 2.8; namely, if a proposition P is provable in the system $\mathcal{N}_i^{\Rightarrow,\wedge,\vee,\perp}$ (or $\mathcal{NG}_i^{\Rightarrow,\wedge,\vee,\perp}$), then it is valid in every Kripke model; that is, it is intuitionistically valid.

Problem 2.38. Prove that $b = \log_2 9$ is irrational. Then, prove that $a = \sqrt{2}$ and $b = \log_2 9$ are two irrational numbers such that a^b is rational.

Problem 2.39. (1) Prove that if $\forall x \neg(P \wedge Q)$ can be deduced intuitionistically from a set of premises Γ, then $\forall x(P \Rightarrow \neg Q)$ and $\forall x(Q \Rightarrow \neg P)$ can also be deduced intuitionistically from Γ.

(2) Prove that if $\forall x(P \vee Q)$ can be deduced intuitionistically from a set of premises Γ, then $\forall x(\neg P \Rightarrow Q)$ and $\forall x(\neg Q \Rightarrow P)$ can also be deduced intuitionistically from Γ.

Conclude that if

$$\forall x(P \vee Q) \quad \text{and} \quad \forall x \neg(P \wedge Q)$$

can be deduced intuitionistically from a set of premises Γ, then

$$\forall x(P \equiv \neg Q) \quad \text{and} \quad \forall x(Q \equiv \neg P)$$

can also be deduced intuitionistically from Γ.

(3) Prove that if $\forall x(P \Rightarrow Q)$ can be deduced intuitionistically from a set of premises Γ, then $\forall x(\neg Q \Rightarrow \neg P)$ can also be deduced intuitionistically from Γ. Use this to prove that if

$$\forall x(P \equiv \neg Q) \quad \text{and} \quad \forall x(Q \equiv \neg P)$$

can be deduced intuitionistically from a set of premises Γ, then $\forall x(\neg\neg P \equiv P)$ and $\forall x(\neg\neg Q \equiv Q)$ can be deduced intuitionistically from Γ.

Problem 2.40. Prove that the formula,

$$\forall x\, \text{even}(2 * x),$$

is provable in Peano arithmetic. Prove that

$$\text{even}(2 * (n + 1) * (n + 3)),$$

is provable in Peano arithmetic for any natural number n.

Problem 2.41. A first-order formula A is said to be in *prenex-form* if either

(1) A is a quantifier-free formula.

(2) $A = \forall t B$ or $A = \exists t B$, where B is in prenex-form.

In other words, a formula is in prenex form iff it is of the form

$$Q_1 t_1 Q_2 t_2 \cdots Q_m t_m \, P,$$

where P is quantifier-free and where $Q_1 Q_2 \cdots Q_m$ is a string of quantifiers, $Q_i \in \{\forall, \exists\}$.

Prove that every first-order formula A is classically equivalent to a formula B in prenex form.

Problem 2.42. Even though natural deduction proof systems for classical propositional logic are complete (with respect to the truth value semantics), they are not adequate for designing algorithms searching for proofs (because of the amount of nondeterminism involved).

Gentzen designed a different kind of proof system using *sequents* (later refined by Kleene, Smullyan, and others) that is far better suited for the design of automated theorem provers. Using such a proof system (a *sequent calculus*), it is relatively easy to design a procedure that terminates for all input propositions P and either certifies that P is (classically) valid or else returns some (or all) falsifying truth assignment(s) for P. In fact, if P is valid, the tree returned by the algorithm can be viewed as a proof of P in this proof system.

For this miniproject, we describe a *Gentzen sequent-calculus* G' for propositional logic that lends itself well to the implementation of algorithms searching for proofs or falsifying truth assignments of propositions.

Such algorithms build trees whose nodes are labeled with pairs of sets called sequents. A *sequent* is a pair of sets of propositions denoted by

$$P_1, \ldots, P_m \to Q_1, \ldots, Q_n,$$

with $m, n \geq 0$. Symbolically, a sequent is usally denoted $\Gamma \to \Delta$, where Γ and Δ are two finite sets of propositions (not necessarily disjoint).

For example,

$$\to P \Rightarrow (Q \Rightarrow P), \;\; P \vee Q \to, \;\; P, Q \to P \wedge Q$$

are sequents. The sequent $\to$, where both $\Gamma = \Delta = \emptyset$ corresponds to falsity.

The choice of the symbol $\to$ to separate the two sets of propositions Γ and Δ is commonly used and was introduced by Gentzen but there is nothing special about it. If you don't like it, you may replace it by any symbol of your choice as long as that symbol does not clash with the logical connectives $(\Rightarrow, \wedge, \vee, \neg)$. For example, you could denote a sequent

$$P_1, \ldots, P_m; Q_1, \ldots, Q_n,$$

using the semicolon as a separator.

Given a truth assignment v to the propositional letters in the propositions P_i and Q_j, we say that v *satisfies the sequent*, $P_1, \ldots, P_m \to Q_1, \ldots, Q_n$, iff

$$v((P_1 \wedge \cdots \wedge P_m) \Rightarrow (Q_1 \vee \cdots \vee Q_n)) = \mathbf{true},$$

or equivalently, v *falsifies the sequent*, $P_1, \ldots, P_m \to Q_1, \ldots, Q_n$, iff

$$v(P_1 \wedge \cdots \wedge P_m \wedge \neg Q_1 \wedge \cdots \wedge \neg Q_n) = \mathbf{true},$$

iff
$$v(P_i) = \mathbf{true},\ 1 \leq i \leq m \quad \text{and} \quad v(Q_j) = \mathbf{false},\ 1 \leq j \leq n.$$
A sequent is *valid* iff it is satisfied by all truth assignments iff it cannot be falsified.

Note that a sequent $P_1, \ldots, P_m \to Q_1, \ldots, Q_n$ can be falsified iff some truth assignment satisfies all of $P_1, \ldots, P_m$ and falsifies all of $Q_1, \ldots, Q_n$. In particular, if $\{P_1, \ldots, P_m\}$ and $\{Q_1, \ldots, Q_n\}$ have some common proposition (they have a nonempty intersection), then the sequent, $P_1, \ldots, P_m \to Q_1, \ldots, Q_n$, is valid. On the other hand if all the P_is and Q_js are propositional letters and $\{P_1, \ldots, P_m\}$ and $\{Q_1, \ldots, Q_n\}$ are disjoint (they have no symbol in common), then the sequent, $P_1, \ldots, P_m \to Q_1, \ldots, Q_n$, is falsified by the truth assignment v where $v(P_i) = \mathbf{true}$, for $i = 1, \ldots m$ and $v(Q_j) = \mathbf{false}$, for $j = 1, \ldots, n$.

The main idea behind the design of the proof system G' is to systematically *try to falsify a sequent*. If such an attempt fails, the sequent is valid and a proof tree is found. Otherwise, all falsifying truth assignments are returned. In some sense

failure to falsify is success (in finding a proof).

The rules of G' are designed so that the conclusion of a rule is falsified by a truth assignment v iff its single premise of one of its two premises is falsified by v. Thus, these rules can be viewed as *two-way* rules that can either be read bottom-up or top-down.

Here are the axioms and the rules of the *sequent calculus* G':

Axioms: $\Gamma, P \to P, \Delta$

Inference rules:

$$\frac{\Gamma, P, Q, \Delta \to \Lambda}{\Gamma, P \land Q, \Delta \to \Lambda} \quad \land\text{: left} \qquad\qquad \frac{\Gamma \to \Delta, P, \Lambda \quad \Gamma \to \Delta, Q, \Lambda}{\Gamma \to \Delta, P \land Q, \Lambda} \quad \land\text{: right}$$

$$\frac{\Gamma, P, \Delta \to \Lambda \quad \Gamma, Q, \Delta \to \Lambda}{\Gamma, P \lor Q, \Delta \to \Lambda} \quad \lor\text{: left} \qquad\qquad \frac{\Gamma \to \Delta, P, Q, \Lambda}{\Gamma \to \Delta, P \lor Q, \Lambda} \quad \lor\text{: right}$$

$$\frac{\Gamma, \Delta \to P, \Lambda \quad Q, \Gamma, \Delta \to \Lambda}{\Gamma, P \Rightarrow Q, \Delta \to \Lambda} \quad \Rightarrow\text{: left} \qquad\qquad \frac{P, \Gamma \to Q, \Delta, \Lambda}{\Gamma \to \Delta, P \Rightarrow Q, \Lambda} \quad \Rightarrow\text{: right}$$

$$\frac{\Gamma, \Delta \to P, \Lambda}{\Gamma, \neg P, \Delta \to \Lambda} \quad \neg\text{: left} \qquad\qquad \frac{P, \Gamma \to \Delta, \Lambda}{\Gamma \to \Delta, \neg P, \Lambda} \quad \neg\text{: right}$$

where Γ, Δ, Λ are any finite sets of propositions, possibly the empty set.

A *deduction tree* is either a one-node tree labeled with a sequent or a tree constructed according to the rules of system G'. A *proof tree* (or *proof*) is a deduction tree whose leaves are *all* axioms. A proof tree for a proposition P is a proof tree for the sequent $\to P$ (with an empty left-hand side).

For example,

$$P, Q \to P$$

is a proof tree.

Here is a proof tree for $(P \Rightarrow Q) \Rightarrow (\neg Q \Rightarrow \neg P)$:

$$\dfrac{\dfrac{\dfrac{P, \neg Q \to P}{\neg Q \to \neg P, P}}{\to P, (\neg Q \Rightarrow \neg P)} \qquad \dfrac{\dfrac{Q \to Q, \neg P}{\neg Q, Q \to \neg P}}{Q \to (\neg Q \Rightarrow \neg P)}}{\dfrac{(P \Rightarrow Q) \to (\neg Q \Rightarrow \neg P)}{\to (P \Rightarrow Q) \Rightarrow (\neg Q \Rightarrow \neg P)}}$$

The following is a deduction tree but not a proof tree,

$$\dfrac{\dfrac{\dfrac{P, R \to P}{R \to \neg P, P}}{\to P, (R \Rightarrow \neg P)} \qquad \dfrac{\dfrac{R, Q, P \to}{R, Q \to \neg P}}{Q \to (R \Rightarrow \neg P)}}{\dfrac{(P \Rightarrow Q) \to (R \Rightarrow \neg P)}{\to (P \Rightarrow Q) \Rightarrow (R \Rightarrow \neg P)}}$$

because its rightmost leaf, $R, Q, P \to$, is falsified by the truth assignment $v(P) = v(Q) = v(R) = \mathbf{true}$, which also falsifies $(P \Rightarrow Q) \Rightarrow (R \Rightarrow \neg P)$.

Let us call a sequent $P_1, \ldots, P_m \to Q_1, \ldots, Q_n$ *finished* if either it is an axiom ($P_i = Q_j$ for some i and some j) or all the propositions P_i and Q_j are atomic and $\{P_1, \ldots, P_m\} \cap \{Q_1, \ldots, Q_n\} = \emptyset$. We also say that a deduction tree is finished if all its leaves are finished sequents.

The beauty of the system G' is that for every sequent, $P_1, \ldots, P_m \to Q_1, \ldots, Q_n$, the process of building a deduction tree from this sequent *always terminates with a tree where all leaves are finished independently of the order in which the rules are applied*. Therefore, we can apply any strategy we want when we build a deduction tree and we are sure that we will get a deduction tree with all its leaves finished. If all the leaves are axioms, then we have a proof tree and the sequent is valid, or else all the leaves that are not axioms yield a falsifying assignment, and all falsifying assignments for the root sequent are found this way.

If we only want to know whether a proposition (or a sequent) is valid, we can stop as soon as we find a finished sequent that is not an axiom because in this case, the input sequent is falsifiable.

(1) Prove that for every sequent $P_1, \ldots, P_m \to Q_1, \ldots, Q_n$ any sequence of applications of the rules of G' terminates with a deduction tree whose leaves are all finished sequents (a finished deduction tree).

Hint. Define the number of connectives $c(P)$ in a proposition P as follows.

(1) If P is a propositional symbol, then

$$c(P) = 0.$$

(2) If $P = \neg Q$, then

$$c(\neg Q) = c(Q) + 1.$$

(3) If $P = Q * R$, where $* \in \{\Rightarrow, \vee, \wedge\}$, then

$$c(Q * R) = c(Q) + c(R) + 1.$$

Given a sequent,

$$\Gamma \to \Delta = P_1, \ldots, P_m \to Q_1, \ldots, Q_n,$$

define the number of connectives, $c(\Gamma \to \Delta)$, in $\Gamma \to \Delta$ by

$$c(\Gamma \to \Delta) = c(P_1) + \cdots + c(P_m) + c(Q_1) + \cdots + c(Q_n).$$

Prove that the application of every rule decreases the number of connectives in the premise(s) of the rule.

(2) Prove that for every sequent $P_1, \ldots, P_m \to Q_1, \ldots, Q_n$ for every finished deduction tree T constructed from $P_1, \ldots, P_m \to Q_1, \ldots, Q_n$ using the rules of G', every truth assignment v satisfies $P_1, \ldots, P_m \to Q_1, \ldots, Q_n$ iff v satisfies every leaf of T. Equivalently, a truth assignment v falsifies $P_1, \ldots, P_m \to Q_1, \ldots, Q_n$ iff v falsifies some leaf of T.

Deduce from the above that a sequent is valid iff all leaves of every finished deduction tree T are axioms. Furthermore, if a sequent is not valid, then for every finished deduction tree T, for that sequent, every falsifying assignment for that sequent is a falsifying assignment of some leaf of the tree, T.

(3) **Programming Project**:
Design an algorithm taking any sequent as input and constructing a finished deduction tree. If the deduction tree is a proof tree, output this proof tree in some fashion (such a tree can be quite big so you may have to find ways of "flattening" these trees). If the sequent is falsifiable, stop when the algorithm encounters the first leaf that is not an axiom and output the corresponding falsifying truth assignment.

I suggest using a *depth-first expansion strategy* for constructing a deduction tree. What this means is that when building a deduction tree, the algorithm will proceed recursively as follows. Given a nonfinished sequent

$$A_1, \ldots, A_p \to B_1, \ldots, B_q,$$

if A_i is the *leftmost* nonatomic proposition if such proposition occurs on the left or if B_j is the leftmost nonatomic proposition if all the A_is are atomic, then

(1) The sequent is of the form

$$\Gamma, A_i, \Delta \to \Lambda,$$

with A_i the leftmost nonatomic proposition. Then either

(a) $A_i = C_i \wedge D_i$ or $A_i = \neg C_i$, in which case either we recursively construct a (finished) deduction tree

$$\begin{array}{c}\mathcal{D}_1\\ \Gamma, C_i, D_i, \Delta \to \Lambda\end{array}$$

to get the deduction tree

$$\dfrac{\begin{array}{c}\mathcal{D}_1\\ \Gamma, C_i, D_i, \Delta \to \Lambda\end{array}}{\Gamma, C_i \wedge D_i, \Delta \to \Lambda}$$

or we recursively construct a (finished) deduction tree

$$\begin{array}{c}\mathcal{D}_1\\ \Gamma, \Delta \to C_i, \Lambda\end{array}$$

to get the deduction tree

$$\dfrac{\begin{array}{c}\mathcal{D}_1\\ \Gamma, \Delta \to C_i, \Lambda\end{array}}{\Gamma, \neg C_i, \Delta \to \Lambda}$$

or

(b) $A_i = C_i \vee D_i$ or $A_i = C_i \Rightarrow D_i$, in which case either we recursively construct two (finished) deduction trees

$$\begin{array}{c}\mathcal{D}_1\\ \Gamma, C_i, \Delta \to \Lambda\end{array} \quad \text{and} \quad \begin{array}{c}\mathcal{D}_2\\ \Gamma, D_i, \Delta \to \Lambda\end{array}$$

to get the deduction tree

$$\dfrac{\begin{array}{c}\mathcal{D}_1\\ \Gamma, C_i, \Delta \to \Lambda\end{array} \qquad\qquad \begin{array}{c}\mathcal{D}_2\\ \Gamma, D_i, \Delta \to \Lambda\end{array}}{\Gamma, C_i \vee D_i, \Delta \to \Lambda}$$

or we recursively construct two (finished) deduction trees

$$\begin{array}{c}\mathcal{D}_1\\ \Gamma, \Delta \to C_i, \Lambda\end{array} \quad \text{and} \quad \begin{array}{c}\mathcal{D}_2\\ D_i, \Gamma, \Delta \to \Lambda\end{array}$$

to get the deduction tree

$$\dfrac{\begin{array}{c}\mathcal{D}_1\\ \Gamma, \Delta \to C_i, \Lambda\end{array} \qquad\qquad \begin{array}{c}\mathcal{D}_2\\ D_i, \Gamma, \Delta \to \Lambda\end{array}}{\Gamma, C_i \Rightarrow D_i, \Delta \to \Lambda}$$

(2) The nonfinished sequent is of the form

$$\Gamma \to \Delta, B_j, \Lambda,$$

with B_j the leftmost nonatomic proposition. Then either

(a) $B_j = C_j \vee D_j$ or $B_j = C_j \Rightarrow D_j$, or $B_j = \neg C_j$, in which case either we recursively construct a (finished) deduction tree

$$\begin{array}{c} \mathcal{D}_1 \\ \Gamma \to \Delta, C_j, D_j, \Lambda \end{array}$$

to get the deduction tree

$$\dfrac{\begin{array}{c} \mathcal{D}_1 \\ \Gamma \to \Delta, C_j, D_j, \Lambda \end{array}}{\Gamma \to \Delta, C_j \vee D_j, \Lambda}$$

or we recursively construct a (finished) deduction tree

$$\begin{array}{c} \mathcal{D}_1 \\ C_j, \Gamma \to D_j, \Delta, \Lambda \end{array}$$

to get the deduction tree

$$\dfrac{\begin{array}{c} \mathcal{D}_1 \\ C_j, \Gamma \to D_j, \Delta, \Lambda \end{array}}{\Gamma \to \Delta, C_j \Rightarrow D_j, \Lambda}$$

or we recursively construct a (finished) deduction tree

$$\begin{array}{c} \mathcal{D}_1 \\ C_j, \Gamma \to \Delta, \Lambda \end{array}$$

to get the deduction tree

$$\dfrac{\begin{array}{c} \mathcal{D}_1 \\ C_j, \Gamma \to \Delta, \Lambda \end{array}}{\Gamma \to \Delta, \neg C_j, \Lambda}$$

or

(b) $B_j = C_j \wedge D_j$, in which case we recursively construct two (finished) deduction trees

$$\begin{array}{c} \mathcal{D}_1 \\ \Gamma \to \Delta, C_j, \Lambda \end{array} \quad \text{and} \quad \begin{array}{c} \mathcal{D}_2 \\ \Gamma \to \Delta, D_j, \Lambda \end{array}$$

to get the deduction tree

$$\dfrac{\begin{array}{cc} \mathcal{D}_1 & \mathcal{D}_2 \\ \Gamma \to \Delta, C_j, \Lambda & \Gamma \to \Delta, D_j, \Lambda \end{array}}{\Gamma \to \Delta, C_j \wedge D_j, \Lambda}$$

If you prefer, you can apply a *breadth-first expansion strategy* for constructing a deduction tree.

Problem 2.43. Let A and be B be any two sets of sets.

(1) Prove that

$$\left(\bigcup A\right) \cup \left(\bigcup B\right) = \bigcup (A \cup B).$$

(2) Assume that A and B are nonempty. Prove that

$$\left(\bigcap A\right) \cap \left(\bigcap B\right) = \bigcap (A \cup B).$$

(3) Assume that A and B are nonempty. Prove that

$$\bigcup (A \cap B) \subseteq \left(\bigcup A\right) \cap \left(\bigcup B\right)$$

and give a counterexample of the inclusion

$$\left(\bigcup A\right) \cap \left(\bigcup B\right) \subseteq \bigcup (A \cap B).$$

Hint. Reduce the above questions to the provability of certain formulae that you have already proved in a previous assignment (you need **not** re-prove these formulae).

Problem 2.44. A set A is said to be *transitive* iff for all $a \in A$ and all $x \in a$, then $x \in A$, or equivalently, for all $a \in A$,

$$a \in A \Rightarrow a \subseteq A.$$

(1) Check that a set A is transitive iff

$$\bigcup A \subseteq A$$

iff

$$A \subseteq 2^A.$$

(2) Recall the definition of the von Neumann successor of a set A given by

$$A^+ = A \cup \{A\}.$$

Prove that if A is a transitive set, then

$$\bigcup (A^+) = A.$$

(3) Recall the von Neumann definition of the natural numbers. Check that for every natural number m

$$m \in m^+ \text{ and } m \subseteq m^+.$$

Prove that every natural number is a transitive set.
Hint. Use induction.

(4) Prove that for any two von Neumann natural numbers m and n, if $m^+ = n^+$, then $m = n$.

(5) Prove that the set, $\mathbb{N}$, of natural numbers is a transitive set.
Hint. Use induction.

Chapter 3

RAM Programs, Turing Machines, and the Partial Computable Functions

See the scanned version of this chapter found in the web page for CIS511:

`http://www.cis.upenn.edu/~jean/old511/html/tcbookpdf3a.pdf`

In this chapter we address the fundamental question

What is a computable function?

Nowadays computers are so pervasive that such a question may seem trivial. Isn't the answer that a function is computable if we can write a program computing it!

This is basically the answer so what more can be said that will shed more light on the question?

The first issue is that we should be more careful about the kind of functions that we are considering. Are we restricting ourselves to total functions or are we allowing partial functions that may not be defined for some of their inputs? It turns out that if we consider functions computed by programs, then partial functions must be considered. In fact, we will see that "deciding" whether a program terminates for all inputs is impossible. But what does deciding mean?

To be mathematically precise requires a fair amount of work. One of the key technical points is the ability to design a program U that takes other programs P as input, and then executes P on any input x. In particular, U should be able to take U itself as input!

Of course a compiler does exactly the above task. But fully describing a compiler for a "real" programming language such as JAVA, PYTHON, C++, *etc.* is a complicated and lengthy task. So a simpler (still quite complicated) way to proceed is to develop a toy programming language and a toy computation model (some kind of machine) capable of executing programs written in our toy language. Then we show how programs in this toy language can be coded so that they can be given as input to other programs. Having done this we need to demonstrate that our language has *universal computing power*. This means that we need to show that a "real" program, say written in JAVA, could be translated into a possibly much longer program written in our toy language. This step is typically an act of faith, in the sense that the details that such a translation can be performed are usually not provided.

A way to be precise regarding universal computing power is to define mathematically a family of functions that should be regarded as "obviously computable," and then to show that the functions computed by the programs written either in our toy programming language or in any modern progamming language are members of this mathematically defined family of computable functions. This step is usually technically very involved, because one needs to show that executing the instructions of a program can be mimicked by functions in our family of computable functions. Conversely, we should prove that every computable function in this family is indeed computable by a program written in our toy programming language or in any modern progamming language. Then we will be have the assurance that we have captured the notion of universal computing power.

Remarkably, Herbrand, Gödel, and Kleene defined such a family of functions in 1934-1935. This is a family of numerical functions $f\colon \mathbb{N}^m \to \mathbb{N}$ containing a subset of very simple functions called base functions, and this family is the smallest family containing the base functions closed under three operations:

1. Composition

2. Primitive recursion

3. Minimization.

Historically, the first two models of computation are the λ-*calculus* of Church (1935) and the *Turing machine* (1936) of Turing. Kleene proved that the λ-definable functions are exactly the (total) computable functions in the sense of Herbrand–Gödel–Kleene in 1936, and Turing proved that the functions computed by Turing machines are exactly the computable functions in the sense of Herbrand–Gödel–Kleene in 1937. Therefore, the λ-calculus and Turing machines have the same "computing power," and both compute exactly the class of computable functions in the sense of Herbrand–Gödel–Kleene. In those days these results were considered quite surprising because the formalism of the λ-calculus has basically nothing to do with the formalism of Turing machines.

Once again we should be more precise about the kinds of functions that we are dealing with. Until Turing (1936), only numerical functions $f\colon \mathbb{N}^m \to \mathbb{N}$ were considered. In order to compute numerical functions in the λ-calculus, Church had to encode the natural numbers are certain λ-terms, which can be viewed as iterators.

Turing assumes that what he calls his *a-machines* (for automatic machines) make use of the symbols 0 and 1 for the purpose of input and output, and if the machine stops, then the output is a *string* of 0s and 1s. Thus a Turing machine can be viewed as computing a function $f\colon (\{0,1\}^*)^m \to \{0,1\}^*$ *on strings*. By allowing a more general alphabet Σ, we see that a Turing machine computes a function $f\colon (\Sigma^*)^m \to \Sigma^*$ on strings over Σ.

At first glance it appears that Turing machines compute a larger class of functions, but this is not so because there exist mutually invertible computable coding functions $C\colon \Sigma^* \to \mathbb{N}$ and decoding functions $D\colon \mathbb{N} \to \Sigma^*$. Using these coding and decoding functions, it suffices to consider numerical functions.

However, Turing machines can also very naturally be viewed as devices for defining computable languages in terms of acceptance and rejection; some kinds of generalized DFA's of NFA's. In this role, it would be very awkward to limit ourselves to sets of natural numbers, although this is possible in theory.

We should also point out that the notion of computable language can be handled in terms of a computation model for functions by considering the characteristic functions of languages. Indeed, a language A is computable (we say decidable) iff its characteristic function χ_A is computable.

The above considerations motivate the definition of the computable functions in the sense of Herbrand–Gödel–Kleene to functions $f\colon (\Sigma^*)^m \to \Sigma^*$ operating *on strings*. However, it is technically simpler to work out all the undecidabilty results for numerical functions of for subsets of $\mathbb{N}$. Since there is no loss of generally in doing so in view of the computable bijections $C\colon \Sigma^* \to \mathbb{N}$ and $D\colon \mathbb{N} \to \Sigma^*$, we will do so.

Nevertherless, in order to deal with languages, it is important to develop a fair amount of computabilty theory about functions computing on strings, so we will present another computation model, the *RAM program model*, which computes functions defined on strings. This model was introduced around 1963 (although it was introduced earlier by Post in a different format). It has the advantage of being closer to actual computer architecture, because the RAM model consists of programs operating on a fixed set of registers. This model is equivalent to the Turing machine model, and the translations, although tedious, are not that bad.

The RAM program model also has the technical advantage that coding up a RAM program as a natural number is not that complicated.

The λ-calculus is a very elegant model but it is more abstract than the RAM program model and the Turing machine model so we postpone discussing it until Chapter 6.

Another very interesting computation model particularly well suited to deal with decidable sets of natural numbers is *Diophantine definability*. This model, arising from the work involved in proving that Hilbert's tenth problem is undecidable, will be discussed in Chapter 8.

In the following sections we will define the RAM program model, the Turing machine model, and then argue without proofs (relegated to an appendix) that there are algorithms to convert RAM programs into Turing machines, and conversely. Then we define the class of computable functions in the sense of Herbrand–Gödel–Kleene, both for numerical functions (defined on $\mathbb{N}$) and functions defined on strings. This will require explaining what is primitive recursion, which is a restricted form of recursion which guarantees that if it is applied to total functions, then the resulting function is total. Intuitively, primitive recursion corresponds to writing programs that only use for loops (loops where the number of iterations is known ahead of time and fixed).

3.1 Partial Functions and RAM Programs

In this section we define an abstract machine model for computing functions

$$f\colon \underbrace{\Sigma^* \times \cdots \times \Sigma^*}_{n} \to \Sigma^*,$$

where $\Sigma = \{a_1, \ldots, a_k\}$ is some input alphabet.

Numerical functions $f\colon \mathbb{N}^n \to \mathbb{N}$ can be viewed as functions defined over the one-letter alphabet $\{a_1\}$, using the bijection $m \mapsto a_1^m$.

Since programs are not guaranteed to terminate for all inputs, we are forced to deal with partial functions so we recall their definition.

Definition 3.1. A binary relation $R \subseteq A \times B$ between two sets A and B is *functional* iff, for all $x \in A$ and $y, z \in B$,

$$(x, y) \in R \quad \text{and} \quad (x, z) \in R \quad \text{implies that} \quad y = z.$$

A *partial function* is a triple $f = \langle A, G, B\rangle$, where A and B are arbitrary sets (possibly empty) and G is a functional relation (possibly empty) between A and B, called the *graph* of f.

Hence, a partial function is a functional relation such that every argument has *at most one image* under f.

The graph of a function f is denoted as $graph(f)$. When no confusion can arise, a function f and its graph are usually identified.

A partial function $f = \langle A, G, B\rangle$ is often denoted as $f\colon A \to B$.

The *domain $dom(f)$* of a partial function $f = \langle A, G, B\rangle$ is the set

$$dom(f) = \{x \in A \mid \exists y \in B,\, (x, y) \in G\}.$$

For every element $x \in dom(f)$, the unique element $y \in B$ such that $(x, y) \in graph(f)$ is denoted as $f(x)$. We say that *$f(x)$ is defined*, also denoted as $f(x)\downarrow$.

If $x \in A$ and $x \notin dom(f)$, we say that *$f(x)$ is undefined*, also denoted as $f(x)\uparrow$.

Intuitively, if a function is partial, it does not return any output for any input not in its domain. This corresponds to an *infinite computation*. It is important to note that two partial functions $f\colon A \to B$ and $f'\colon A' \to B'$ are equal iff $A = A'$, $B = B'$, and for all $a \in A$, either both $f(a)$ and $f'(a)$ are defined and $f(a) = f'(a)$, or both $f(a)$ and $f'(a)$ are undefined.

A partial function $f\colon A \to B$ is a *total function* iff $dom(f) = A$. *It is customary to call a total function simply a function.*

We now define a model of computation know as the *RAM programs* or *Post machines*.

RAM programs are written in a sort of assembly language involving simple instructions manipulating strings stored into registers.

Every RAM program uses a fixed and finite number of *registers* denoted as $R1, \ldots, Rp$, with no limitation on the size of strings held in the registers.

RAM programs can be defined either in flowchart form or in linear form. Since the linear form is more convenient for coding purposes, we present RAM programs in linear form.

A RAM program P (in linear form) consists of a finite sequence of *instructions* using a finite number of registers $R1, \ldots, Rp$.

Instructions may optionally be labeled with line numbers denoted as $N1, \ldots, Nq$.

It is neither mandatory to label all instructions, nor to use distinct line numbers! Thus the same line number can be used in more than one line. As we will see later on, this makes it easier to concatenate two different programs without performing a renumbering of line numbers.

Every instruction has *four fields*, not necessarily all used. The main field is the **op-code**. Here is an example of a RAM program to concatenate two strings x_1 and x_2.

$$
\begin{array}{llcl}
 & R3 & \leftarrow & R1 \\
 & R4 & \leftarrow & R2 \\
N0 & R4 & \mathtt{jmp}_a & N1b \\
 & R4 & \mathtt{jmp}_b & N2b \\
 & & \mathtt{jmp} & N3b \\
N1 & & \mathtt{add}_a & R3 \\
 & & \mathtt{tail} & R4 \\
 & & \mathtt{jmp} & N0a \\
N2 & & \mathtt{add}_b & R3 \\
 & & \mathtt{tail} & R4 \\
 & & \mathtt{jmp} & N0a \\
N3 & R1 & \leftarrow & R3 \\
 & & \mathtt{continue} &
\end{array}
$$

Definition 3.2. *RAM programs* are constructed from seven types of *instructions* shown below:

(1_j)	N		`add`$_j$	Y
(2)	N		`tail`	Y
(3)	N		`clr`	Y
(4)	N	Y	$\leftarrow$	X
$(5a)$	N		`jmp`	$N1a$
$(5b)$	N		`jmp`	$N1b$
(6_ja)	N	Y	`jmp`$_j$	$N1a$
(6_jb)	N	Y	`jmp`$_j$	$N1b$
(7)	N		`continue`	

1. An instruction of type (1_j) concatenates the letter a_j to the right of the string held by register Y $(1 \leq j \leq k)$. The effect is the assignment
$$Y := Ya_j.$$

2. An instruction of type (2) deletes the leftmost letter of the string held by the register Y. This corresponds to the function *tail*, defined such that
$$\begin{aligned} tail(\epsilon) &= \epsilon, \\ tail(a_ju) &= u. \end{aligned}$$
The effect is the assignment
$$Y := tail(Y).$$

3. An instruction of type (3) clears register Y, i.e., sets its value to the empty string ϵ. The effect is the assignment
$$Y := \epsilon.$$

4. An instruction of type (4) assigns the value of register X to register Y. The effect is the assignment
$$Y := X.$$

5. An instruction of type (5a) or (5b) is an unconditional jump.

   The effect of (5a) is to jump to the closest line number $N1$ occurring above the instruction being executed, and the effect of (5b) is to jump to the closest line number $N1$ occurring below the instruction being executed.

6. An instruction of type (6_ja) or (6_jb) is a conditional jump. Let *head* be the function defined as follows:
$$\begin{aligned} head(\epsilon) &= \epsilon, \\ head(a_ju) &= a_j. \end{aligned}$$

The effect of $(6_j a)$ is to jump to the closest line number $N1$ occurring above the instruction being executed iff $head(Y) = a_j$, else to execute the next instruction (the one immediately following the instruction being executed).

The effect of $(6_j b)$ is to jump to the closest line number $N1$ occurring below the instruction being executed iff $head(Y) = a_j$, else to execute the next instruction.

When computing over $\mathbb{N}$, instructions of type $(6_j a)$ or $(6_j b)$ jump to the closest $N1$ above or below iff Y is nonnull.

7. An instruction of type (7) is a no-op, i.e., the registers are unaffected. If there is a next instruction, then it is executed, else, the program stops.

Obviously, a program is syntactically correct only if certain conditions hold.

Definition 3.3. A *RAM program* P is a finite sequence of instructions as in Definition 3.2, and satisfying the following conditions:

(1) For every jump instruction (conditional or not), the line number to be jumped to must exist in P.

(2) The last instruction of a RAM program is a `continue`.

The reason for allowing multiple occurences of line numbers is to make it easier to concatenate programs without having to perform a renaming of line numbers.

The technical choice of jumping to the closest address $N1$ above or below comes from the fact that it is easy to search up or down using primitive recursion, as we will see later on.

For the purpose of computing a function $f\colon \underbrace{\Sigma^* \times \cdots \times \Sigma^*}_{n} \to \Sigma^*$ using a RAM program P, we assume that P has at least n registers called *input registers*, and that these registers $R1, \ldots, Rn$ are initialized with the input values of the function f. We also assume that *the output is returned in register* $R1$.

Example 3.1. The following RAM program concatenates two strings x_1 and x_2 held in registers $R1$ and $R2$. Since $\Sigma = \{a, b\}$, for more clarity, we wrote $\mathtt{jmp}_a$ instead of $\mathtt{jmp}_1$, $\mathtt{jmp}_b$ instead of $\mathtt{jmp}_2$, $\mathtt{add}_a$ instead of $\mathtt{add}_1$, and $\mathtt{add}_b$ instead of $\mathtt{add}_2$.

	$R3$	$\leftarrow$	$R1$
	$R4$	$\leftarrow$	$R2$
$N0$	$R4$	$\mathtt{jmp}_a$	$N1b$
	$R4$	$\mathtt{jmp}_b$	$N2b$
		`jmp`	$N3b$
$N1$		$\mathtt{add}_a$	$R3$
		`tail`	$R4$
		`jmp`	$N0a$
$N2$		$\mathtt{add}_b$	$R3$
		`tail`	$R4$
		`jmp`	$N0a$
$N3$	$R1$	$\leftarrow$	$R3$
		`continue`	

Definition 3.4. A RAM program P *computes the partial function* $\varphi\colon (\Sigma^*)^n \to \Sigma^*$ if the following conditions hold: For every input $(x_1, \ldots, x_n) \in (\Sigma^*)^n$, having initialized the input registers $R1, \ldots, Rn$ with $x_1, \ldots, x_n$, the program eventually halts iff $\varphi(x_1, \ldots, x_n)$ is defined, and if and when P halts, the value of $R1$ is equal to $\varphi(x_1, \ldots, x_n)$. A partial function φ is *RAM-computable* iff it is computed by some RAM program.

For example, the following program computes the *erase function E* defined such that

$$E(u) = \epsilon$$

for all $u \in \Sigma^*$:

`clr`	$R1$
`continue`	

The following program computes the *jth successor function S_j* defined such that

$$S_j(u) = ua_j$$

for all $u \in \Sigma^*$:

$\mathtt{add}_j$	$R1$
`continue`	

The following program (with n input variables) computes the *projection function P_i^n* defined such that

$$P_i^n(u_1, \ldots, u_n) = u_i,$$

where $n \geq 1$, and $1 \leq i \leq n$:

$R1$	$\leftarrow$	Ri
	`continue`	

Note that P_1^1 is the identity function.

Having a programming language, we would like to know how powerful it is, that is, we would like to know what kind of functions are RAM-computable. At first glance, it seems that RAM programs don't do much, but this is not so. Indeed, we will see shortly that the class of RAM-computable functions is quite extensive.

One way of getting new programs from previous ones is via composition. Another one is by primitive recursion. We will investigate these constructions after introducing another model of computation, *Turing machines*.

Remarkably, the classes of (partial) functions computed by RAM programs and by Turing machines are identical. This is the class of *partial computable functions* in the sense of Herbrand–Gödel–Kleene, also called *partial recursive functions*, a term which is now considered old-fashion. We will present the definition of the so-called *μ-recursive functions* (due to Kleene).

The following proposition will be needed to simplify the encoding of RAM programs as numbers.

Proposition 3.1. *Every RAM program can be converted to an equivalent program only using the following type of instructions:*

$$\begin{array}{llll}
(1_j) & N & & \texttt{add}_j \quad Y \\
(2) & N & & \texttt{tail} \quad Y \\
(6_ja) & N & Y & \texttt{jmp}_j \quad N1a \\
(6_jb) & N & Y & \texttt{jmp}_j \quad N1b \\
(7) & N & & \texttt{continue}
\end{array}$$

The proof is fairly simple. For example, instructions of the form

$$Ri \leftarrow Rj$$

can be eliminated by transferring the contents of Rj into an auxiliary register Rk, and then by transferring the contents of Rk into Ri and Rj.

3.2 Definition of a Turing Machine

We define a Turing machine model for computing functions

$$f\colon \underbrace{\Sigma^* \times \cdots \times \Sigma^*}_{n} \to \Sigma^*,$$

where $\Sigma = \{a_1, \ldots, a_k\}$ is some input alphabet. In this section, since we are primarily interested in computing functions we only consider deterministic Turing machines.

There are many variants of the Turing machine model. The main decision that needs to be made has to do with the kind of tape used by the machine. We opt for a single *finite* *tape* that is both an input and a storage mechanism. This tape can be viewed as a string over *tape alphabet* Γ such that $\Sigma \subseteq \Gamma$. There is a read/write head pointing to some symbol on the tape, symbols on the tape can be overwritten, and the read/write head can move one symbol to the left or one symbol to the right, also causing a state transition. When the write/read head attempts to move past the rightmost or the leftmost symbol on the tape, the tape is allowed to grow. To accomodate such a move, the tape alphabet contains some special symbol $B \notin \Sigma$, the *blank*, and this symbol is added to the tape as the new leftmost or rightmost symbol on the tape.

A common variant uses a tape which is infinite at both ends, but only has finitely many symbols not equal to B, so effectively it is equivalent to a finite tape allowed to grow at either ends. Another variant uses a semi-infinite tape infinite to the right, but with a left end. We find this model cumbersome because it requires shifting right the entire tape when a left move is attempted from the left end of the tape.

Another decision that needs to be made is the format of the instructions. Does an instruction cause *both* a state transition *and* a symbol overwrite, or do we have separate instructions for a state transition and a symbol overwrite. In the first case, an instruction can be specified as a quintuple, and in the second case by a quadruple. We opt for quintuples. Here is our definition.

Definition 3.5. A (deterministic) *Turing machine* (or *TM*) M is a sextuple $M = (K, \Sigma, \Gamma, \{L, R\}, \delta, q_0)$, where

- K is a finite set of *states*;
- Σ is a finite *input alphabet*;
- Γ is a finite *tape alphabet*, s.t. $\Sigma \subseteq \Gamma$, $K \cap \Gamma = \emptyset$, and with blank $B \notin \Sigma$;
- $q_0 \in K$ is the *start state* (or *initial state*);
- δ is the *transition function*, a (finite) set of quintuples

$$\delta \subseteq K \times \Gamma \times \Gamma \times \{L, R\} \times K,$$

  such that for all $(p, a) \in K \times \Gamma$, there is at most one triple $(b, m, q) \in \Gamma \times \{L, R\} \times K$ such that $(p, a, b, m, q) \in \delta$.

A quintuple $(p, a, b, m, q) \in \delta$ is called an *instruction*. It is also denoted as

$$p, a \to b, m, q.$$

The effect of an instruction is to switch from state p to state q, overwrite the symbol currently scanned a with b, and move the read/write head either left or right, according to m.

Example 3.2. Here is an example of a Turing machine specified by

$K = \{q_0, q_1, q_2, q_3\}$; $\Sigma = \{a, b\}$; $\Gamma = \{a, b, B\}$;

The instructions in δ are:

$$\begin{aligned}
q_0, B &\to B, R, q_3, \\
q_0, a &\to b, R, q_1, \\
q_0, b &\to a, R, q_1, \\
q_1, a &\to b, R, q_1, \\
q_1, b &\to a, R, q_1, \\
q_1, B &\to B, L, q_2, \\
q_2, a &\to a, L, q_2, \\
q_2, b &\to b, L, q_2, \\
q_2, B &\to B, R, q_3.
\end{aligned}$$

3.3 Computations of Turing Machines

To explain how a Turing machine works, we describe its action on *instantaneous descriptions*. We take advantage of the fact that $K \cap \Gamma = \emptyset$ to define instantaneous descriptions.

Definition 3.6. Given a Turing machine

$$M = (K, \Sigma, \Gamma, \{L, R\}, \delta, q_0),$$

an *instantaneous description* (for short an *ID*) is a (nonempty) string in $\Gamma^* K \Gamma^+$, that is, a string of the form

$$upav,$$

where $u, v \in \Gamma^*$, $p \in K$, and $a \in \Gamma$.

The intuition is that an ID $upav$ describes a snapshot of a TM in the current state p, whose tape contains the string uav, and with the read/write head pointing to the symbol a. Thus, in $upav$, the state p is just to the left of the symbol presently scanned by the read/write head.

We explain how a TM works by showing how it acts on ID's.

Definition 3.7. Given a Turing machine

$$M = (K, \Sigma, \Gamma, \{L, R\}, \delta, q_0),$$

the *yield relation (or compute relation)* $\vdash$ is a binary relation defined on the set of ID's as follows. For any two ID's ID_1 and ID_2, we have $ID_1 \vdash ID_2$ iff either

(1) $(p, a, b, R, q) \in \delta$, and either

   (a) $ID_1 = upacv$, $c \in \Gamma$, and $ID_2 = ubqcv$, or

   (b) $ID_1 = upa$ and $ID_2 = ubqB$;

or

(2) $(p, a, b, L, q) \in \delta$, and either

   (a) $ID_1 = ucpav$, $c \in \Gamma$, and $ID_2 = uqcbv$, or

   (b) $ID_1 = pav$ and $ID_2 = qBbv$.

Note how the tape is extended by one blank after the rightmost symbol in case (1)(b), and by one blank before the leftmost symbol in case (2)(b).

As usual, we let $\vdash^+$ denote the transitive closure of $\vdash$, and we let $\vdash^*$ denote the reflexive and transitive closure of $\vdash$. We can now explain how a Turing machine computes a partial function

$$f\colon \underbrace{\Sigma^* \times \cdots \times \Sigma^*}_{n} \to \Sigma^*.$$

Since we allow functions taking $n \geq 1$ input strings, we assume that Γ contains the special delimiter **,** not in Σ, used to separate the various input strings.

It is convenient to assume that a Turing machine "cleans up" its tape when it halts before returning its output. What this means is that when the Turing machine halts, the output should be clearly identifiable, so all symbols not in $\Sigma \cup \{B\}$ that may have been used during the computation must be erased. Thus when the TM stops the tape must consist of a string $w \in \Sigma^*$ possibly surrounded by blanks (the symbol B). Actually, if the output is ϵ, the tape must contain a nonempty string of blanks. To achieve this technically, we define proper ID's.

Definition 3.8. Given a Turing machine

$$M = (K, \Sigma, \Gamma, \{L, R\}, \delta, q_0),$$

where Γ contains some delimiter **,** not in Σ in addition to the blank B, a *starting ID* is of the form

$$q_0 w_1,w_2,\ldots,w_n$$

where $w_1, \ldots, w_n \in \Sigma^*$ and $n \geq 2$, or $q_0 w$ with $w \in \Sigma^+$, or $q_0 B$.

A *blocking (or halting) ID* is an ID $upav$ such that there are no instructions $(p, a, b, m, q) \in \delta$ for any $(b, m, q) \in \Gamma \times \{L, R\} \times K$.

A *proper ID* is a halting ID of the form

$$B^h p w B^l,$$

where $w \in \Sigma^*$, and $h, l \geq 0$ (with $l \geq 1$ when $w = \epsilon$).

Computation sequences are defined as follows.

Definition 3.9. Given a Turing machine

$$M = (K, \Sigma, \Gamma, \{L, R\}, \delta, q_0),$$

a *computation sequence (or computation)* is a finite or infinite sequence of ID's

$$ID_0, ID_1, \ldots, ID_i, ID_{i+1}, \ldots,$$

such that $ID_i \vdash ID_{i+1}$ for all $i \geq 0$.

A computation sequence *halts* iff it is a finite sequence of ID's, so that

$$ID_0 \vdash^* ID_n,$$

and ID_n is a halting ID.

A computation sequence *diverges* if it is an infinite sequence of ID's.

We now explain how a Turing machine computes a partial function.

Definition 3.10. A Turing machine

$$M = (K, \Sigma, \Gamma, \{L, R\}, \delta, q_0)$$

computes the partial function

$$f\colon \underbrace{\Sigma^* \times \cdots \times \Sigma^*}_{n} \to \Sigma^*$$

iff the following conditions hold:

(1) For every $w_1, \ldots, w_n \in \Sigma^*$, given the starting ID

$$ID_0 = q_0 w_1{,}w_2, \ldots {,}w_n$$

or $q_0 w$ with $w \in \Sigma^+$, or $q_0 B$, the computation sequence of M from ID_0 halts in a proper ID iff $f(w_1, \ldots, w_n)$ is defined.

(2) If $f(w_1, \ldots, w_n)$ is defined, then M halts in a proper ID of the form

$$ID_n = B^h p f(w_1, \ldots, w_n) B^l,$$

which means that it computes the right value.

A function f (over Σ^*) is *Turing computable* iff it is computed by some Turing machine M.

Note that by (1), the TM M may halt in an improper ID, in which case $f(w_1, \ldots, w_n)$ must be undefined. This corresponds to the fact that *we only accept to retrieve the output of a computation if the TM has cleaned up its tape*, *i.e.*, produced a proper ID. In particular, intermediate calculations have to be erased before halting.

Example 3.3. Consider the Turing machine specified by $K = \{q_0, q_1, q_2, q_3\}$; $\Sigma = \{a, b\}$; $\Gamma = \{a, b, B\}$;

The instructions in δ are:

$$
\begin{aligned}
q_0, B &\to B, R, q_3, \\
q_0, a &\to b, R, q_1, \\
q_0, b &\to a, R, q_1, \\
q_1, a &\to b, R, q_1, \\
q_1, b &\to a, R, q_1, \\
q_1, B &\to B, L, q_2, \\
q_2, a &\to a, L, q_2, \\
q_2, b &\to b, L, q_2, \\
q_2, B &\to B, R, q_3.
\end{aligned}
$$

The reader can easily verify that this machine exchanges the a's and b's in a string. For example, on input $w = aaababb$, the output is $bbbabaa$.

3.4 Equivalence of RAM Programs And Turing Machines

Turing machines can simulate RAM programs, and as a result, we have the following Theorem.

Theorem 3.2. *Every RAM-computable function is Turing-computable. Furthermore, given a RAM program P, we can effectively construct a Turing machine M computing the same function.*

The idea of the proof is to represent the contents of the registers $R1, \ldots Rp$ on the Turing machine tape by the string

$$\#r1\#r2\#\cdots\#rp\#,$$

where # is a special marker and ri represents the string held by Ri, We also use Proposition 3.1 to reduce the number of instructions to be dealt with.

The Turing machine M is built of blocks, each block simulating the effect of some instruction of the program P. The details are a bit tedious, and can be found in an appendix or in Machtey and Young [36].

RAM programs can also simulate Turing machines.

Theorem 3.3. *Every Turing-computable function is RAM-computable. Furthermore, given a Turing machine M, one can effectively construct a RAM program P computing the same function.*

The idea of the proof is to design a RAM program containing an encoding of the current ID of the Turing machine M in register $R1$, and to use other registers $R2, R3$ to simulate the effect of executing an instruction of M by updating the ID of M in $R1$.

The details are tedious and can be found in ann appendix.

Another proof can be obtained by proving that the class of Turing computable functions coincides with the class of *partial computable functions* (formerly called *partial recursive functions*), to be defined shortly. Indeed, it turns out that both RAM programs and Turing machines compute precisely the class of partial recursive functions. For this, we will need to define the *primitive recursive functions*.

Informally, a primitive recursive function is a total recursive function that can be computed using only **for** loops, that is, loops in which the number of iterations is fixed (unlike a **while** loop). A formal definition of the primitive functions is given in Section 3.7. For the time being we make the following provisional definition.

Definition 3.11. Let $\Sigma = \{a_1, \ldots, a_k\}$. The class of *partial computable functions* also called *partial recursive functions* is the class of partial functions (over Σ^*) that can be computed by RAM programs (or equivalently by Turing machines).

The class of *computable functions* also called *recursive functions* is the subset of the class of partial computable functions consisting of functions defined for every input (i.e., total functions).

Turing machines can also be used as acceptors to define languages so we introduce the basic relevant definitions. A more detailed study of these languages will be provided in Chapter 5.

3.5 Listable Languages and Computable Languages

We define the computably enumerable languages, also called listable languages, and the computable languages. The old-fashion terminology for listable languages is recursively enumerable languages, and for computable languages is recursive languages.

When operating as an acceptor, a Turing machine takes a single string as input and either goes on forever or halts with the answer "accept" or "reject." One way to deal with acceptance or rejection is to assume that the TM has a set of final states. Another way more consistent with our view that machines compute functions is to assume that the TM's under consideration have a tape alphabet containing the special symbols 0 and 1. Then acceptance is signaled by the output 1, and rejection is signaled by the output 0.

Note that with our convention that in order to produce an output a TM must halt in a proper ID, the TM must erase the tape before outputing 0 or 1.

Definition 3.12. Let $\Sigma = \{a_1, \ldots, a_k\}$. A language $L \subseteq \Sigma^*$ is *(Turing) listable* or *(Turing) computably enumerable (for short, a c.e. set)* (or *recursively enumerable (for short, a r.e. set)*) iff there is some TM M such that for every $w \in L$, M halts in a proper ID with the output 1, and for every $w \notin L$, either M halts in a proper ID with the output 0 or it runs forever.

A language $L \subseteq \Sigma^*$ is *(Turing) computable* (or *recursive*) iff there is some TM M such that for every $w \in L$, M halts in a proper ID with the output 1, and for every $w \notin L$, M halts in a proper ID with the output 0.

Thus, given a computably enumerable language L, for some $w \notin L$, it is possible that a TM accepting L runs forever on input w. On the other hand, for a computable (recursive) language L, a TM accepting L always halts in a proper ID.

When dealing with languages, it is often useful to consider *nondeterministic Turing machines*. Such machines are defined just like deterministic Turing machines, except that their transition function δ is just a (finite) set of quintuples

$$\delta \subseteq K \times \Gamma \times \Gamma \times \{L, R\} \times K,$$

with no particular extra condition.

It can be shown that every nondeterministic Turing machine can be simulated by a deterministic Turing machine, and thus, nondeterministic Turing machines also accept the class of c.e. sets. This is a very tedious simulation, and very few books actually provide all the details!

It can be shown that a computably enumerable language is the range of some computable (recursive) function; see Section 5.4. It can also be shown that a language L is computable (recursive) iff both L and its complement are computably enumerable; see Section 5.4. There are computably enumerable languages that are not computable (recursive); see Section 5.4.

3.6 A Simple Function Not Known to be Computable

The "$3n + 1$ problem" proposed by Collatz around 1937 is the following:

Given any positive integer $n \geq 1$, construct the sequence $c_i(n)$ as follows starting with $i = 1$:

$$
\begin{aligned}
c_1(n) &= n \\
c_{i+1}(n) &= \begin{cases} c_i(n)/2 & \text{if } c_i(n) \text{ is even} \\ 3c_i(n) + 1 & \text{if } c_i(n) \text{ is odd.} \end{cases}
\end{aligned}
$$

Observe that for $n = 1$, we get the infinite periodic sequence

$$1 \Longrightarrow 4 \Longrightarrow 2 \Longrightarrow 1 \Longrightarrow 4 \Longrightarrow 2 \Longrightarrow 1 \Longrightarrow \cdots,$$

so we may assume that we stop the first time that the sequence $c_i(n)$ reaches the value 1 (if it actually does). Such an index i is called the *stopping time* of the sequence. And this is the problem:

Conjecture (Collatz):

For any starting integer value $n \geq 1$, the sequence $(c_i(n))$ always reaches 1.

Starting with $n = 3$, we get the sequence

$$3 \Longrightarrow 10 \Longrightarrow 5 \Longrightarrow 16 \Longrightarrow 8 \Longrightarrow 4 \Longrightarrow 2 \Longrightarrow 1.$$

Starting with $n = 5$, we get the sequence

$$5 \Longrightarrow 16 \Longrightarrow 8 \Longrightarrow 4 \Longrightarrow 2 \Longrightarrow 1.$$

Starting with $n = 6$, we get the sequence

$$6 \Longrightarrow 3 \Longrightarrow 10 \Longrightarrow 5 \Longrightarrow 16 \Longrightarrow 8 \Longrightarrow 4 \Longrightarrow 2 \Longrightarrow 1.$$

Starting with $n = 7$, we get the sequence

$$
\begin{aligned}
&7 \Longrightarrow 22 \Longrightarrow 11 \Longrightarrow 34 \Longrightarrow 17 \Longrightarrow 52 \Longrightarrow 26 \Longrightarrow 13 \Longrightarrow 40 \\
&\Longrightarrow 20 \Longrightarrow 10 \Longrightarrow 25 \Longrightarrow 16 \Longrightarrow 8 \Longrightarrow 4 \Longrightarrow 2 \Longrightarrow 1.
\end{aligned}
$$

One might be surprised to find that for $n = 27$, it takes 111 steps to reach 1, and for $n = 97$, it takes 118 steps. I computed the stopping times for n up to 10^7 and found that the largest stopping time, 686 (685 steps) is obtained for $n = 8400511$. The terms of this sequence reach values over 1.5×10^{11}. The graph of the sequence $c(8400511)$ is shown in Figure 3.1.

We can define the partial computable function C (with positive integer inputs) defined by

$$C(n) = \text{the smallest } i \text{ such that } c_i(n) = 1 \text{ if it exists.}$$

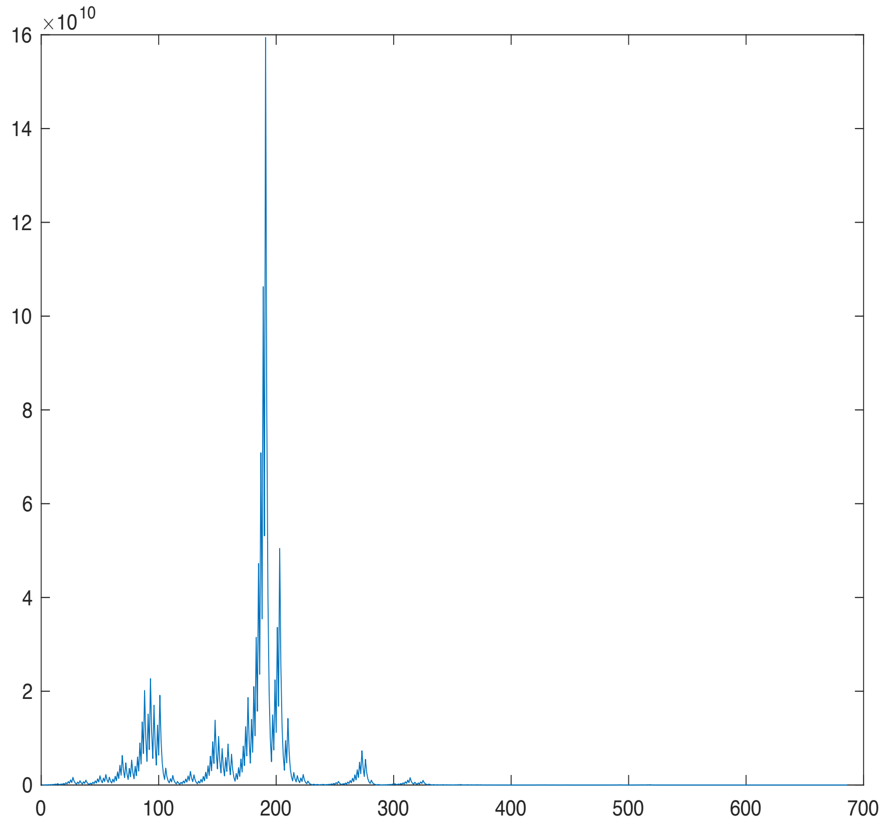


Figure 3.1: Graph of the sequence for $n = 8400511$.

Then the Collatz conjecture is equivalent to asserting that the function C is (total) computable. The graph of the function C for $1 \leq n \leq 10^7$ is shown in Figure 3.2.

So far, the conjecture remains open. It has been checked by computer for all integers less than or equal to 87×2^{60}.

We now return to the computability of functions. Our goal is to define the partial computable functions in the sense of Herbrand–Gödel–Kleene. This class of functions is defined from some base functions in terms of three closure operations:

1. Composition

2. Primitive recursion

3. Minimization.

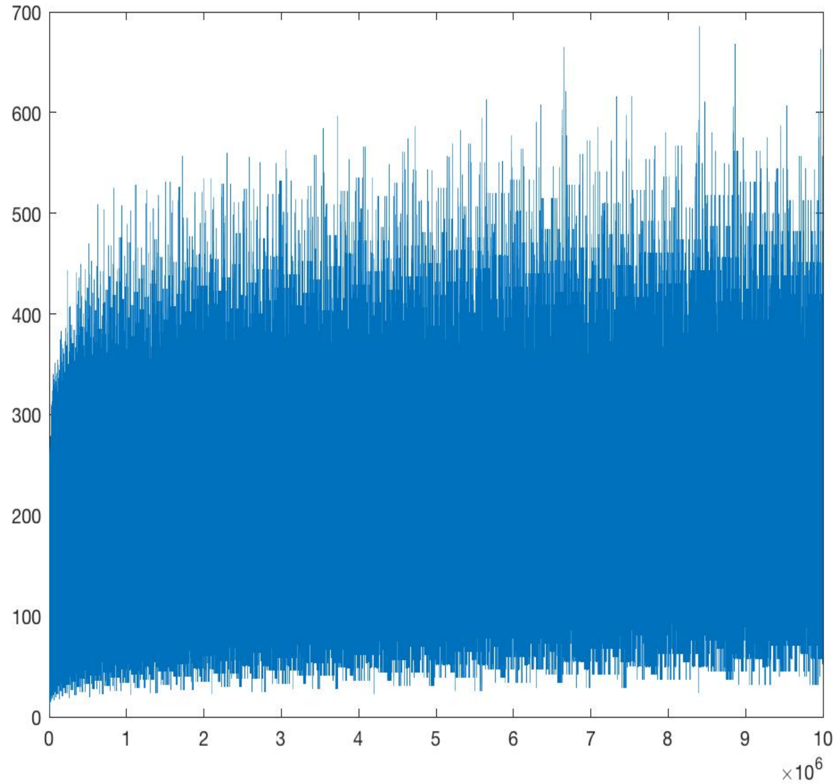


Figure 3.2: Graph of the function C for $1 \leq n \leq 10^7$.

The first two operations preserve the property of a function to be total, and this subclass of total computable functions called *primitive recursive functions* plays an important technical role.

3.7 The Primitive Recursive Functions

Historically the primitive recursive functions were defined for numerical functions (computing on the natural numbers). Since one of our goals is to show that the RAM-computable functions are partial recursive, we define the primitive recursive functions as functions $f\colon (\Sigma^*)^m \to \Sigma^*$, where $\Sigma = \{a_1, \ldots, a_k\}$ is a finite alphabet. As usual, by assuming that $\Sigma = \{a_1\}$, we can deal with numerical functions $f\colon \mathbb{N}^m \to \mathbb{N}$.

The class of primitive recursive functions is defined in terms of base functions and two closure operations.

Definition 3.13. Let $\Sigma = \{a_1, \ldots, a_k\}$. The *base functions* over Σ are the following functions:

(1) The *erase function E*, defined such that $E(w) = \epsilon$, for all $w \in \Sigma^*$;

(2) For every j, $1 \leq j \leq k$, the *j-successor function S_j*, defined such that $S_j(w) = wa_j$, for all $w \in \Sigma^*$;

(3) The *projection functions P_i^n*, defined such that

$$P_i^n(w_1, \ldots, w_n) = w_i,$$

for every $n \geq 1$, every i, $1 \leq i \leq n$, and for all $w_1, \ldots, w_n \in \Sigma^*$.

Note that P_1^1 is the identity function on Σ^*. Projection functions can be used to permute, duplicate, or drop the arguments of another function.

In the special case where we are only considering numerical functions ($\Sigma = \{a_1\}$), the function $E\colon \mathbb{N} \to \mathbb{N}$ is the *zero function* given by $E(n) = 0$ for all $n \in \mathbb{Z}$, and it is often denoted by Z. There is a single successor function $S_{a_1}\colon \mathbb{N} \to \mathbb{N}$ usually denoted S (or **Succ**) given by $S(n) = n + 1$ for all $n \in \mathbb{N}$.

Even though in this section we are primarily interested in total functions, later on, the same closure operations will be applied to partial functions so we state the definition of the closure operations in the more general case of partial functions. The first closure operation is (extended) composition.

Definition 3.14. Let $\Sigma = \{a_1, \ldots, a_k\}$. For any partial or total function

$$g\colon \underbrace{\Sigma^* \times \cdots \times \Sigma^*}_{m} \to \Sigma^*,$$

and any $m \geq 1$ partial or total functions

$$h_i\colon \underbrace{\Sigma^* \times \cdots \times \Sigma^*}_{n} \to \Sigma^*, \quad n \geq 1,$$

the *composition of g and the h_i* is the partial function

$$f\colon \underbrace{\Sigma^* \times \cdots \times \Sigma^*}_{n} \to \Sigma^*,$$

denoted as $g \circ (h_1, \ldots, h_m)$, such that

$$f(w_1, \ldots, w_n) = g(h_1(w_1, \ldots, w_n), \ldots, h_m(w_1, \ldots, w_n)),$$

for all $w_1, \ldots, w_n \in \Sigma^*$. If g and all the h_i are total functions, then $g \circ (h_1, \ldots, h_m)$ is obviously a total function. But if g or any of the h_i is a partial function, then the value *$(g \circ (h_1, \ldots, h_m))(x_1, \ldots, x_n)$ is defined if and only if all the values $h_i(x_1, \ldots, x_n)$ are defined for $i = 1, \ldots, m$, and $g(h_1(w_1, \ldots, w_n), \ldots, h_m(w_1, \ldots, w_n))$ is defined.*

Thus even if g "ignores" some of its inputs, in computing $g(h_1(w_1,\ldots,w_n),\ldots,h_m(w_1,\ldots,w_n))$, all arguments $h_i(w_1,\ldots,w_n)$ *must* be evaluated.

As an example of a composition, $f = g \circ (P_2^2, P_1^2)$ is such that

$$f(w_1,w_2) = g(P_2^2(w_1,w_2), P_1^2(w_1,w_2)) = g(w_2,w_1).$$

The second closure operation is *primitive recursion*. First we define primitive recursion for numerical functions because it is simpler.

Definition 3.15. Given any two partial or total functions $g\colon \mathbb{N}^{m-1} \to \mathbb{N}$ and $h\colon \mathbb{N}^{m+1} \to \mathbb{N}$ ($m \geq 2$), the partial or total function $f\colon \mathbb{N}^m \to \mathbb{N}$ is defined by *primitive recursion from g and h* if f is given by

$$\begin{aligned} f(0,x_2,\ldots,x_m) &= g(x_2,\ldots,x_m), \\ f(n+1,x_2,\ldots,x_m) &= h(n, f(n,x_2,\ldots,x_m), x_2,\ldots,x_m), \end{aligned}$$

for all $n, x_2,\ldots,x_m \in \mathbb{N}$. When $m = 1$, we have

$$\begin{aligned} f(0) &= b, \\ f(n+1) &= h(n, f(n)), \quad \text{for all } n \in \mathbb{N}, \end{aligned}$$

for some fixed natural number $b \in \mathbb{N}$.

If g and h are total functions, it is easy to show that f is also a total function. If g or h is partial, obviously $f(0,x_2,\ldots,x_m)$ is defined iff $g(x_2,\ldots,x_m)$ is defined, and $f(n+1,x_2,\ldots,x_m)$ is defined iff $f(n,x_2,\ldots,x_m)$ is defined and $h(n, f(n,x_2,\ldots,x_m), x_2,\ldots,x_m)$ is defined.

Definition 3.15 is quite a straightjacket in the sense that $n+1$ must be the first argument of f, and the definition only applies if h has $m+1$ arguments, but in practice a "natural" definition often ignores the argument n and some of the arguments $x_2,\ldots,x_m$. This is where the projection functions come into play to drop, duplicate, or permute arguments.

For example, a "natural" definition of the predecessor function *pred* is

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

but this is not a legal primitive recursive definition. To make it a legal primitive recursive definition we need the function $h = P_1^2$, and a legal primitive recursive definition for *pred* is

$$\begin{aligned} \mathit{pred}(0) &= 0 \\ \mathit{pred}(m+1) &= P_1^2(m, \mathit{pred}(m)). \end{aligned}$$

Addition, multiplication, exponentiation, and super-exponentiation, can be defined by primitive recursion as follows (being a bit loose, for supexp we should use some projections ...):

$$\begin{aligned}
add(0, n) &= P_1^1(n) = n, \\
add(m+1, n) &= S \circ P_2^3(m, add(m, n), n) \\
&= S(add(m, n)) \\
mult(0, n) &= E(n) = 0, \\
mult(m+1, n) &= add \circ (P_2^3, P_3^3)(m, mult(m, n), n) \\
&= add(mult(m, n), n), \\
rexp(0, n) &= S \circ E(n) = 1, \\
rexp(m+1, n) &= mult(rexp(m, n), n), \\
exp(m, n) &= rexp \circ (P_2^2, P_1^2)(m, n), \\
supexp(0, n) &= 1, \\
supexp(m+1, n) &= exp(n, supexp(m, n)).
\end{aligned}$$

We usually write $m + n$ for $add(m, n)$, $m * n$ or even mn for $mult(m, n)$, and m^n for $exp(m, n)$.

There is a minus operation on $\mathbb{N}$ named *monus*. This operation denoted by $\dot{-}$ is defined by

$$m \dot{-} n = \begin{cases} m - n & \text{if } m \geq n \\ 0 & \text{if } m < n. \end{cases}$$

Then *monus* is defined by

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

except that the above is not a legal primitive recursion. For one thing, recursion should be performed on m, not n. We can define *rmonus* as

$$rmonus(n, m) = m \dot{-} n,$$

and then $m \dot{-} n = (rmonus \circ (P_2^2, P_1^2))(m, n)$, and

$$\begin{aligned}
rmonus(0 \dot{-} m) &= P_1^1(m) \\
rmonus(n+1, m) &= pred \circ P_2^2(n, rmonus(n, m)).
\end{aligned}$$

The following functions are also primitive recursive:

$$sg(n) = \begin{cases} 1 & \text{if } n > 0 \\ 0 & \text{if } n = 0, \end{cases}$$

$$\overline{sg}(n) = \begin{cases} 0 & \text{if } n > 0 \\ 1 & \text{if } n = 0, \end{cases}$$

as well as

$$abs(m, n) = |m - m| = m \dot{-} n + n \dot{-} m,$$

and

$$eq(m, n) = \begin{cases} 1 & \text{if } m = n \\ 0 & \text{if } m \neq n. \end{cases}$$

Indeed

$$\begin{aligned} sg(0) &= 0 \\ sg(n + 1) &= S \circ E \circ P_1^2(n, sg(n)) \\ \overline{sg}(n) &= S(E(n)) \dot{-} sg(n) = 1 \dot{-} sg(n), \end{aligned}$$

and

$$eq(m, n) = sg(|m - n|).$$

Finally, the function

$$cond(m, n, p, q) = \begin{cases} p & \text{if } m = n \\ q & \text{if } m \neq n, \end{cases}$$

is primitive recursive since

$$cond(m, n, p, q) = eq(m, n) * p + \overline{sg}(eq(m, n)) * q.$$

We can also design more general version of *cond*. For example, define $compare_{\leq}$ as

$$compare_{\leq}(m, n) = \begin{cases} 1 & \text{if } m \leq n \\ 0 & \text{if } m > n, \end{cases}$$

which is given by

$$compare_{\leq}(m, n) = 1 \dot{-} sg(m \dot{-} n).$$

Then we can define

$$cond_{\leq}(m, n, p, q) = \begin{cases} p & \text{if } m \leq n \\ q & \text{if } m > n, \end{cases}$$

with

$$cond_{\leq}(m, n, n, p) = compare_{\leq}(m, n) * p + \overline{sg}(compare_{\leq}(m, n)) * q.$$

The above allows to define functions by cases.

We now generalize primitive recursion to functions defined on strings (in Σ^*). The new twist is that instead of the argument $n+1$ of f, we need to consider the k arguments ua_i of f for $i = 1, \ldots, k$ (with $u \in \Sigma^*$), so instead of a single function h, we need k functions h_i to define primitive recursively what $f(ua_i, w_2, \ldots, w_m)$ is.

Definition 3.16. Let $\Sigma = \{a_1, \ldots, a_k\}$. For any partial or total function

$$g\colon \underbrace{\Sigma^* \times \cdots \times \Sigma^*}_{m-1} \to \Sigma^*,$$

where $m \geq 2$, and any k partial or total functions

$$h_i\colon \underbrace{\Sigma^* \times \cdots \times \Sigma^*}_{m+1} \to \Sigma^*,$$

the partial function

$$f\colon \underbrace{\Sigma^* \times \cdots \times \Sigma^*}_{m} \to \Sigma^*,$$

is defined by *primitive recursion from g and $h_1, \ldots, h_k$*, if

$$\begin{aligned}
f(\epsilon, w_2, \ldots, w_m) &= g(w_2, \ldots, w_m),\\
f(ua_1, w_2, \ldots, w_m) &= h_1(u, f(u, w_2, \ldots, w_m), w_2, \ldots, w_m),\\
\cdots &= \cdots\\
f(ua_k, w_2, \ldots, w_m) &= h_k(u, f(u, w_2, \ldots, w_m), w_2, \ldots, w_m),
\end{aligned}$$

for all $u, w_2, \ldots, w_m \in \Sigma^*$.

When $m = 1$, for some fixed $w \in \Sigma^*$, we have

$$\begin{aligned}
f(\epsilon) &= w,\\
f(ua_1) &= h_1(u, f(u)),\\
\cdots &= \cdots\\
f(ua_k) &= h_k(u, f(u)),
\end{aligned}$$

for all $u \in \Sigma^*$.

Again, if g and the h_i are total, it is easy to see that f is total.

As an example over $\{a, b\}^*$, the following function $g\colon \Sigma^* \times \Sigma^* \to \Sigma^*$, is defined by primitive recursion:

$$\begin{aligned}
g(\epsilon, v) &= P_1^1(v),\\
g(ua_i, v) &= S_i \circ P_2^3(u, g(u, v), v),
\end{aligned}$$

where $1 \leq i \leq k$. It is easily verified that $g(u, v) = vu$. Then,

$$con = g \circ (P_2^2, P_1^2)$$

computes the concatenation function, *i.e.*, $con(u, v) = uv$. Here are some primitive recursive functions that often appear as building blocks for other primitive recursive functions.

The *delete last* function *dell* given by

$$\begin{aligned} dell(\epsilon) &= \epsilon \\ del(ua_i) &= u, \quad 1 \leq i \leq k,\ u \in \Sigma^* \end{aligned}$$

is defined primitive recursively by

$$\begin{aligned} dell(\epsilon) &= \epsilon \\ dell(ua_i) &= P_1^2(u, dell(u)), \quad 1 \leq i \leq k,\ u \in \Sigma^*. \end{aligned}$$

For every string $w \in \Sigma^*$, the constant function c_w given by

$$c_w(u) = w \quad \text{for all } u \in \Sigma^*$$

is defined primitive recursively by induction on the length of w by

$$\begin{aligned} c_\epsilon &= E \\ c_{va_i} &= S_i \circ c_v, \quad 1 \leq i \leq k. \end{aligned}$$

The *sign function* sg given by

$$sg(x) = \begin{cases} \epsilon & \text{if } x = \epsilon \\ a_1 & \text{if } x \neq \epsilon \end{cases}$$

is defined primitive recursively by

$$\begin{aligned} sg(\epsilon) &= \epsilon \\ sg(ua_i) &= (c_{a_1} \circ P_1^2)(u, sg(u)). \end{aligned}$$

The *anti-sign function* $\overline{sg}$ given by

$$\overline{sg}(x) = \begin{cases} a_1 & \text{if } x = \epsilon \\ \epsilon & \text{if } x \neq \epsilon \end{cases}$$

is primitive recursive. The proof is left an an exercise.

The function end_j $(1 \leq j \leq k)$ given by

$$end_j(x) = \begin{cases} a_1 & \text{if } x \text{ ends with } a_j \\ \epsilon & \text{otherwise} \end{cases}$$

is primitive recursive. The proof is left an an exercise.

The reverse function $rev\colon \Sigma^* \to \Sigma^*$ given by $rev(u) = u^R$ is primitive recursive, because

$$\begin{aligned} rev(\epsilon) &= \epsilon \\ rev(ua_i) &= (con \circ (c_{a_i} \circ P_1^2, P_2^2))(u, rev(u)), \quad 1 \leq i \leq k. \end{aligned}$$

The *tail function* *tail* given by

$$\begin{aligned} tail(\epsilon) &= \epsilon \\ tail(a_i u) &= u \end{aligned}$$

is primitive recursive, because

$$tail = rev \circ dell \circ rev.$$

The *last function* *last* given by

$$\begin{aligned} last(\epsilon) &= \epsilon \\ last(u a_i) &= a_i \end{aligned}$$

is primitive recursive, because

$$\begin{aligned} last(\epsilon) &= \epsilon \\ last(u a_i) &= c_{a_i} \circ P_1^2(u, last(u)). \end{aligned}$$

The *head function* *head* given by

$$\begin{aligned} head(\epsilon) &= \epsilon \\ head(a_i u) &= a_i \end{aligned}$$

is primitive recursive, because

$$head = last \circ rev.$$

We are now ready to define the class of primitive recursive functions.

Definition 3.17. Let $\Sigma = \{a_1, \ldots, a_k\}$. The class of *primitive recursive functions* is the smallest class of (total) functions (over Σ^*) which contains the base functions and is closed under composition and primitive recursion.

In the special where $k = 1$, we obtain the class of *numerical primitive recursive functions*.

The class of primitive recursive functions may not seem very big, but it contains all the total functions that we would ever want to compute. Although it is rather tedious to prove, the following theorem can be shown.

Theorem 3.4. *For any alphabet $\Sigma = \{a_1, \ldots, a_k\}$, every primitive recursive function is RAM computable, and thus Turing computable.*

The proof is given in an appendix.

In order to define new functions it is also useful to use predicates.

3.8 Primitive Recursive Predicates

Primitive recursive predicates will be used in Section 4.3.

Definition 3.18. An *n-ary predicate* P over $\mathbb{N}$ is any subset of $\mathbb{N}^n$. We write that a tuple $(x_1, \ldots, x_n)$ *satisfies* P as $(x_1, \ldots, x_n) \in P$ or as $P(x_1, \ldots, x_n)$. The *characteristic function* of a predicate P is the function $C_P \colon \mathbb{N}^n \to \{0, 1\}$ defined by

$$C_p(x_1, \ldots, x_n) = \begin{cases} 1 & \text{iff } P(x_1, \ldots, x_n) \text{ holds} \\ 0 & \text{iff } \textbf{not } P(x_1, \ldots, x_n). \end{cases}$$

A predicate P (over $\mathbb{N}$) is *primitive recursive* iff its characteristic function C_P is primitive recursive.

More generally, an *n-ary predicate* P (over Σ^*) is any subset of $(\Sigma^*)^n$. We write that a tuple $(x_1, \ldots, x_n)$ *satisfies* P as $(x_1, \ldots, x_n) \in P$ or as $P(x_1, \ldots, x_n)$. The *characteristic function* of a predicate P is the function $C_P \colon (\Sigma^*)^n \to \{a_1\}^*$ defined by

$$C_p(x_1, \ldots, x_n) = \begin{cases} a_1 & \text{iff } P(x_1, \ldots, x_n) \text{ holds} \\ \epsilon & \text{iff } \textbf{not } P(x_1, \ldots, x_n). \end{cases}$$

A predicate P (over Σ^*) is *primitive recursive* iff its characteristic function C_P is primitive recursive.

Since we will only need to use primitive recursive predicates over $\mathbb{N}$ in the following chapters, for simplicity of exposition we will restrict ourselves to such predicates. The general case in treated in Machtey and Young [36].

It is easily shown that if P and Q are primitive recursive predicates (over $(\mathbb{N}^n)$), then $P \vee Q$, $P \wedge Q$ and $\neg P$ are also primitive recursive.

As an exercise, the reader may want to prove that the predicate,

prime(n) iff n is a prime number, is a primitive recursive predicate.

For any fixed $k \geq 1$, the function

ord(k, n) = exponent of the kth prime in the prime factorization of n, is a primitive recursive function.

We can also define functions by cases.

Proposition 3.5. *If $P_1, \ldots, P_n$ are pairwise disjoint primitive recursive predicates (which means that $P_i \cap P_j = \emptyset$ for all $i \neq j$) and $f_1, \ldots, f_{n+1}$ are primitive recursive functions, the function g defined below is also primitive recursive:*

$$g(\overline{x}) = \begin{cases} f_1(\overline{x}) & \textit{iff } P_1(\overline{x}) \\ \vdots & \\ f_n(\overline{x}) & \textit{iff } P_n(\overline{x}) \\ f_{n+1}(\overline{x}) & \textit{otherwise.} \end{cases}$$

Here we write $\overline{x}$ for $(x_1, \ldots, x_n)$.

It is also useful to have bounded quantification and bounded minimization. Recall that we are restricting our attention to numerical predicates and functions, so all variables range over $\mathbb{N}$. Proofs of the results stated below can be found in Machtey and Young [36].

Definition 3.19. If P is an $(n+1)$-ary predicate, then the *bounded existential predicate* $(\exists y \leq x)P(y, \overline{z})$ holds iff some $y \leq x$ makes $P(y, \overline{z})$ true.

The *bounded universal predicate* $(\forall y \leq x)P(y, \overline{z})$ holds iff every $y \leq x$ makes $P(y, \overline{z})$ true.

Proposition 3.6. *If P is an $(n+1)$-ary primitive recursive predicate, then $(\exists y \leq x)P(y, \overline{z})$ and $(\forall y \leq x)P(y, \overline{z})$ are also primitive recursive predicates.*

As an application, we can show that the equality predicate, $u = v?$, is primitive recursive. The following slight generalization of Proposition 3.6 will be needed in Section 4.3.

Proposition 3.7. *If P is an $(n+1)$-ary primitive recursive predicate and $f\colon \mathbb{N}^n \to \mathbb{N}$ is a primitive recursive function, then $(\exists y \leq f(\overline{z}))P(y, \overline{z})$ is also a primitive recursive predicate.*

Definition 3.20. If P is an $(n+1)$-ary predicate, then the *bounded minimization of* P, $\min(y \leq x)\, P(y, \overline{z})$, is the function defined such that $\min(y \leq x)\, P(y, \overline{z})$ is the least natural number $y \leq x$ such that $P(y, \overline{z})$ if such a y exists, $x + 1$ otherwise.

The *bounded maximization of* P, $\max(y \leq x)\, P(y, \overline{z})$, is the function defined such that $\max(y \leq x)\, P(y, \overline{z})$ is the largest natural number $y \leq x$ such that $P(y, \overline{z})$ if such a y exists, $x + 1$ otherwise.

Proposition 3.8. *If P is an $(n+1)$-ary primitive recursive predicate, then $\min(y \leq x)\, P(y, \overline{z})$ and $\max(y \leq x)\, P(y, \overline{z})$ are primitive recursive functions.*

So far the primitive recursive functions do not yield all the Turing-computable functions. The following proposition also shows that restricting ourselves to total functions is too limiting.

Let $\mathcal{F}$ be any set of total functions that contains the base functions and is closed under composition and primitive recursion (and thus, $\mathcal{F}$ contains all the primitive recursive functions).

Definition 3.21. We say that a function $f\colon \Sigma^* \times \Sigma^* \to \Sigma^*$ is *universal* for the one-argument functions in $\mathcal{F}$ iff for every function $g\colon \Sigma^* \to \Sigma^*$ in $\mathcal{F}$, there is some $n \in \mathbb{N}$ such that

$$f(a_1^n, u) = g(u)$$

for all $u \in \Sigma^*$.

Proposition 3.9. *For any countable set $\mathcal{F}$ of total functions containing the base functions and closed under composition and primitive recursion, if f is a universal function for the functions $g\colon \Sigma^* \to \Sigma^*$ in $\mathcal{F}$, then $f \notin \mathcal{F}$.*

Proof. Assume that the universal function f is in $\mathcal{F}$. Let g be the function such that

$$g(u) = f(a_1^{|u|}, u)a_1$$

for all $u \in \Sigma^*$. We claim that $g \in \mathcal{F}$. It is enough to prove that the function h such that

$$h(u) = a_1^{|u|}$$

is primitive recursive, which is easily shown.

Then, because f is universal, there is some m such that

$$g(u) = f(a_1^m, u)$$

for all $u \in \Sigma^*$. Letting $u = a_1^m$, we get

$$g(a_1^m) = f(a_1^m, a_1^m) = f(a_1^m, a_1^m)a_1,$$

a contradiction. $\square$

Thus, either a universal function for $\mathcal{F}$ is partial, or it is not in $\mathcal{F}$.

In order to get a larger class of functions, we need the closure operation known as minimization.

3.9 The Partial Computable Functions

Minimization can be viewed as an abstract version of a while loop. First let us consider the simpler case of numerical functions.

Consider a function $g\colon \mathbb{N}^{m+1} \to \mathbb{N}$, with $m \geq 0$. We would like to know if for any fixed $n_1, \ldots, n_m \in \mathbb{N}$, the equation

$$g(n, n_1, \ldots, n_m) = 0 \quad \text{with respect to } n \in \mathbb{N}$$

has a solution $n \in \mathbb{N}$, and if so, we return the smallest such solution. Thus we are defining a (partial) function $f\colon \mathbb{N}^m \to \mathbb{N}$ such that

$$f(n_1, \ldots, n_m) = \min\{n \in \mathbb{N} \mid g(n, n_1, \ldots, n_m) = 0\},$$

with the understanding that $f(n_1, \ldots, n_m)$ is undefined otherwise. If g is computed by a RAM program, computing $f(n_1, \ldots, n_m)$ corresponds to the while loop

$n := 0$;
while $g(n, n_1, \ldots, n_m) \neq 0$ **do**
$n := n + 1$;
endwhile
let $f(n_1, \ldots, n_m) = n$.

Definition 3.22. For any function $g\colon \mathbb{N}^{m+1} \to \mathbb{N}$, where $m \geq 0$, the function $f\colon \mathbb{N}^m \to \mathbb{N}$ is defined by *minimization from* g, if the following conditions hold for all $n_1, \ldots, n_m \in \mathbb{N}$:

(1) $f(n_1, \ldots, n_m)$ is defined iff there is some $n \in \mathbb{N}$ such that $g(p, n_1, \ldots, n_m)$ is defined for all p, $0 \leq p \leq n$, and

$$g(n, n_1, \ldots, n_m) = 0;$$

(2) When $f(n_1, \ldots, n_m)$ is defined,

$$f(n_1, \ldots, n_m) = n,$$

where n is such that $g(n, n_1, \ldots, n_m) = 0$ and $g(p, n_1, \ldots, n_m) \neq 0$ for every p, $0 \leq p \leq n - 1$. In other words, n is the smallest natural number such that $g(n, n_1, \ldots, n_m) = 0$.

Following Kleene, we write

$$f(n_1, \ldots, n_m) = \mu n[g(n, n_1, \ldots, n_m) = 0].$$

Remark: When $f(n_1, \ldots, n_m)$ is defined, $f(n_1, \ldots, n_m) = n$, where n is the smallest natural number such that condition (1) holds. It is very important to require that all the values $g(p, n_1, \ldots, n_m)$ be defined for all p, $0 \leq p \leq n$, when defining $f(n_1, \ldots, n_m)$. Failure to do so allows non-computable functions.

Minimization can be generalized to functions defined on strings as follows. Given a function $g\colon (\Sigma^*)^{m+1} \to \Sigma^*$, for any fixed $w_1, \ldots, w_m \in \Sigma^*$, we wish to solve the equation

$$g(u, w_1, \ldots, w_m) = \epsilon \quad \text{with respect to } u \in \Sigma^*,$$

and return the "smallest" solution u, if any. The only issue is, what does smallest solution mean. We resolve this issue by restricting u to be a string of a_j's, for some fixed letter $a_j \in \Sigma$. Thus there are k variants of minimization corresponding to searching for a shortest string in $\{a_j\}^*$, for a fixed j, $1 \leq j \leq k$.

Let $\Sigma = \{a_1, \ldots, a_k\}$. For any function

$$g\colon \underbrace{\Sigma^* \times \cdots \times \Sigma^*}_{m+1} \to \Sigma^*,$$

where $m \geq 0$, for every j, $1 \leq j \leq k$, the function

$$f\colon \underbrace{\Sigma^* \times \cdots \times \Sigma^*}_{m} \to \Sigma^*$$

looks for the shortest string u over $\{a_j\}^*$ (for a fixed j) such that

$$g(u, w_1, \ldots, w_m) = \epsilon :$$

This corresponds to the following while loop:

$u := \epsilon$;
while $g(u, w_1, \ldots, w_m) \neq \epsilon$ **do**
$u := ua_j$;
endwhile
let $f(w_1, \ldots, w_m) = u$

The operation of minimization (sometimes called minimalization) is defined as follows.

Definition 3.23. Let $\Sigma = \{a_1, \ldots, a_k\}$. For any function

$$g\colon \underbrace{\Sigma^* \times \cdots \times \Sigma^*}_{m+1} \to \Sigma^*,$$

where $m \geq 0$, for every j, $1 \leq j \leq k$, the function

$$f\colon \underbrace{\Sigma^* \times \cdots \times \Sigma^*}_{m} \to \Sigma^*,$$

is defined by *minimization over* $\{a_j\}^*$ *from* g, if the following conditions hold for all $w_1, \ldots,$ $w_m \in \Sigma^*$:

(1) $f(w_1, \ldots, w_m)$ is defined iff there is some $n \geq 0$ such that $g(a_j^p, w_1, \ldots, w_m)$ is defined for all p, $0 \leq p \leq n$, and
$$g(a_j^n, w_1, \ldots, w_m) = \epsilon.$$

(2) When $f(w_1, \ldots, w_m)$ is defined,
$$f(w_1, \ldots, w_m) = a_j^n,$$
where n is such that
$$g(a_j^n, w_1, \ldots, w_m) = \epsilon$$
and
$$g(a_j^p, w_1, \ldots, w_m) \neq \epsilon$$
for every p, $0 \leq p \leq n-1$.

We write
$$f(w_1, \ldots, w_m) = min_j u[g(u, w_1, \ldots, w_m) = \epsilon].$$

Note: When $f(w_1, \ldots, w_m)$ is defined,
$$f(w_1, \ldots, w_m) = a_j^n,$$
where n is the smallest natural number such that condition (1) holds. It is very important to require that all the values $g(a_j^p, w_1, \ldots, w_m)$ be defined for all p, $0 \leq p \leq n$, when defining $f(w_1, \ldots, w_m)$. Failure to do so allows non-computable functions.

Remark: Inspired by Kleene's notation in the case of numerical functions, we may use the *μ-notation*:

$$f(w_1, \ldots, w_m) = \mu_j u[g(u, w_1, \ldots, w_m) = \epsilon].$$

The class of partial computable functions is defined as follows.

Definition 3.24. Let $\Sigma = \{a_1, \ldots, a_k\}$. The class of *partial computable functions* (in the sense of Herbrand–Gödel–Kleene), also called *partial recursive functions* is the smallest class of partial functions (over Σ^*) which contains the base functions and is closed under composition, primitive recursion, and minimization.

The class of *computable functions* also called *recursive functions* is the subset of the class of partial computable functions consisting of functions defined for every input (i.e., total functions).

One of the major results of computability theory is the following theorem.

Theorem 3.10. *For an alphabet $\Sigma = \{a_1, \ldots, a_k\}$, every partial computable function (partial recursive function) is RAM-computable, and thus Turing-computable. Conversely, every Turing-computable function is a partial computable function (partial recursive function). Similarly, the class of computable functions (recursive functions) is equal to the class of Turing-computable functions that halt in a proper ID for every input.*

First we prove that every partial computable function is RAM-computable, which is not that difficult because composition, primitive recursion, and minimization are easily implemented using RAM programs. By Theorem 3.2, every RAM program can be converted to a Turing machine, so every partial computable function is Turing-computable.

For the converse, one can show that given a Turing machine, there is a primitive recursive function describing how to go from one ID to the next. Then minimization is used to guess whether a computation halts. The proof shows that every partial computable function needs minimization *at most once*. The characterization of the computable functions in terms of TM's follows easily. Details are given in an appendix. See also Machtey and Young [36] and Kleene I.M. [32] (Chapter XIII).

We will prove in Section 4.3 that every RAM-computable function (over $\mathbb{N}$) is partial computable. This will be done by encoding RAM programs as natural numbers.

There are computable functions (recursive functions) that are not primitive recursive. Such an example is given by Ackermann's function.

Ackermann's function is the function $A\colon \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ which is defined by the following recursive clauses:

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

It turns out that A is a computable function which is **not** primitive recursive. This is not easy to prove. It can be shown that:

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

and

$$A(4, x) = 2^{2^{\cdot^{\cdot^{\cdot^{2^{16}}}}}}\Big\}^{x} - 3,$$

with $A(4, 0) = 16 - 3 = 13$.

For example

$$A(4, 1) = 2^{16} - 3, \quad A(4, 2) = 2^{2^{16}} - 3.$$

Actually, it is not so obvious that A is a total function, but it is.

Proposition 3.11. *Ackermann's function A is a total function.*

Proof. This is shown by induction, using the lexicographic ordering $\preceq$ on $\mathbb{N} \times \mathbb{N}$, which is defined as follows:

$$\begin{aligned}&(m, n) \preceq (m', n') \quad \text{iff either}\\ &m = m' \text{ and } n = n', \text{ or}\\ &m < m', \text{ or}\\ &m = m' \text{ and } n < n'.\end{aligned}$$

We write $(m, n) \prec (m', n')$ when $(m, n) \preceq (m', n')$ and $(m, n) \neq (m', n')$.

We prove that $A(m, n)$ is defined for all $(m, n) \in \mathbb{N} \times \mathbb{N}$ by complete induction over the lexicographic ordering on $\mathbb{N} \times \mathbb{N}$.

In the base case, $(m, n) = (0, 0)$, and since $A(0, n) = n + 1$, we have $A(0, 0) = 1$, and $A(0, 0)$ is defined.

For $(m, n) \neq (0, 0)$, the induction hypothesis is that $A(m', n')$ is defined for all $(m', n') \prec (m, n)$. We need to conclude that $A(m, n)$ is defined.

If $m = 0$, since $A(0, n) = n + 1$, $A(0, n)$ is defined.

If $m \neq 0$ and $n = 0$, since

$$(m - 1, 1) \prec (m, 0),$$

by the induction hypothesis, $A(m - 1, 1)$ is defined, but $A(m, 0) = A(m - 1, 1)$, and thus $A(m, 0)$ is defined.

If $m \neq 0$ and $n \neq 0$, since

$$(m, n-1) \prec (m, n),$$

by the induction hypothesis, $A(m, n-1)$ is defined. Since

$$(m-1, A(m, n-1)) \prec (m, n),$$

by the induction hypothesis, $A(m-1, A(m, n-1))$ is defined. But $A(m, n) = A(m-1, A(m, n-1))$, and thus $A(m, n)$ is defined.

Thus, $A(m, n)$ is defined for all $(m, n) \in \mathbb{N} \times \mathbb{N}$. $\square$

It is possible to show that A is a computable (recursive) function, although the quickest way to prove it requires some fancy machinery (the recursion theorem; see Section 7.1). Proving that A is *not* primitive recursive is even harder.

A further study of the partial recursive functions requires the notions of pairing functions and of universal functions (or universal Turing machines).

Chapter 4

Universal RAM Programs and Undecidability of the Halting Problem

The goal of this chapter is to prove three of the main results of computability theory:

(1) The undecidability of the halting problem for RAM programs (and Turing machines).

(2) The existence of universal RAM programs.

(3) The existence of the Kleene T-predicate.

All three require the ability to code a RAM program as a natural number. Gödel pioneered the technique of encoding objects such as proofs as natural numbers in his famous paper on the (first) incompleteness theorem (1931). One of the technical issues is to code (pack) a tuple of natural numbers as a single natural number, so that the numbers being packed can be retrieved. Gödel designed a fancy function whose defintion does not involve recursion (Gödel's β function; see Kleene [32] or Shoenfield [45]). For our purposes, a simpler function J due to Cantor packing two natural numbers m and n as a single natural number $J(m, n)$ suffices.

Another technical issue is the fact it is possible to reduce most of computability theory to numerical functions $f\colon \mathbb{N}^m \to \mathbb{N}$, and even to functions $f\colon \mathbb{N} \to \mathbb{N}$. Indeed, there are primitive recursive coding and decoding functions $D_k\colon \Sigma^* \to \mathbb{N}$ and $C_k\colon \mathbb{N} \to \Sigma^*$ such that $C_k \circ D_k = \mathrm{id}_{\Sigma^*}$, where $\Sigma = \{a_1, \ldots, a_k\}$. It is simpler to code programs (or Turing machines) taking natural numbers as input.

Unfortunately, these coding techniques are very tedious so we advise the reader not to get bogged down with technical details upon first reading.

4.1 Pairing Functions

Pairing functions are used to encode pairs of integers into single integers, or more generally, finite sequences of integers into single integers. We begin by exhibiting a bijective pairing function $J\colon \mathbb{N}^2 \to \mathbb{N}$. The function J has the graph partially showed below:

$$\begin{array}{cccccccccccc}
y & & & & & & & & & & & \\
4 & 10 & & & & & & & & & & \\
 & & \searrow & & & & & & & & & \\
3 & 6 & & 11 & & & & & & & & \\
 & & \searrow & & \searrow & & & & & & & \\
2 & 3 & & 7 & & 12 & & & & & & \\
 & & \searrow & & \searrow & & \searrow & & & & & \\
1 & 1 & & 4 & & 8 & & 13 & & & & \\
 & & \searrow & & \searrow & & \searrow & & \searrow & & & \\
0 & 0 & & 2 & & 5 & & 9 & & 14 & & \\
 & 0 & & 1 & & 2 & & 3 & & 4 & x &
\end{array}$$

The function J corresponds to a certain way of enumerating pairs of integers (x, y). Note that the value of $x + y$ is constant along each descending diagonal, and consequently, we have

$$\begin{aligned}
J(x, y) &= 1 + 2 + \cdots + (x + y) + x, \\
&= ((x + y)(x + y + 1) + 2x)/2, \\
&= ((x + y)^2 + 3x + y)/2,
\end{aligned}$$

that is,

$$J(x, y) = ((x + y)^2 + 3x + y)/2.$$

For example, $J(0, 3) = 6,\ J(1, 2) = 7,\ J(2, 2) = 12,\ J(3, 1) = 13,\ J(4, 0) = 14.$

Let $K\colon \mathbb{N} \to \mathbb{N}$ and $L\colon \mathbb{N} \to \mathbb{N}$ be the projection functions onto the axes, that is, the unique functions such that

$$K(J(a, b)) = a \quad \text{and} \quad L(J(a, b)) = b,$$

for all $a, b \in \mathbb{N}$. For example, $K(11) = 1$, and $L(11) = 3$; $K(12) = 2$, and $L(12) = 2$; $K(13) = 3$ and $L(13) = 1$.

Definition 4.1. The *pairing function* $J\colon \mathbb{N}^2 \to \mathbb{N}$ is defined by

$$J(x, y) = ((x + y)^2 + 3x + y)/2 \quad \text{for all } x, y \in \mathbb{N}.$$

The functions $K\colon \mathbb{N} \to \mathbb{N}$ and $L\colon \mathbb{N} \to \mathbb{N}$ are the *projection functions* onto the axes, that is, the unique functions such that

$$K(J(a, b)) = a \quad \text{and} \quad L(J(a, b)) = b,$$

for all $a, b \in \mathbb{N}$.

The functions J, K, L are called *Cantor's pairing functions*. They were used by Cantor to prove that the set $\mathbb{Q}$ of rational numbers is countable.

Clearly, J is primitive recursive, since it is given by a polynomial. It is not hard to prove that J is injective and surjective, and that it is strictly monotonic in each argument, which means that for all $x, x', y, y' \in \mathbb{N}$, if $x < x'$ then $J(x, y) < J(x', y)$, and if $y < y'$ then $J(x, y) < J(x, y')$.

The projection functions can be computed explicitly, although this is a bit tricky. We only need to observe that by monotonicity of J,

$$x \leq J(x, y) \quad \text{and} \quad y \leq J(x, y),$$

and thus,

$$K(z) = \min(x \leq z)(\exists y \leq z)[J(x, y) = z],$$

and

$$L(z) = \min(y \leq z)(\exists x \leq z)[J(x, y) = z].$$

Therefore, by the results of Section 3.8, K and L are primitive recursive. It can be verified that $J(K(z), L(z)) = z$, for all $z \in \mathbb{N}$.

More explicit formulae can be given for K and L. If we define

$$\begin{aligned} Q_1(z) &= \lfloor (\lfloor \sqrt{8z+1} \rfloor + 1)/2 \rfloor - 1 \\ Q_2(z) &= 2z - (Q_1(z))^2, \end{aligned}$$

then it can be shown that

$$\begin{aligned} K(z) &= \frac{1}{2}(Q_2(z) - Q_1(z)) \\ L(z) &= Q_1(z) - \frac{1}{2}(Q_2(z) - Q_1(z)). \end{aligned}$$

In the above formula, the function $m \mapsto \lfloor \sqrt{m} \rfloor$ yields the largest integer s such that $s^2 \leq m$. It can be computed by a RAM program.

The pairing function $J(x, y)$ is also denoted as $\langle x, y \rangle$, and K and L are also denoted as Π_1 and Π_2. The notation $\langle x, y \rangle$ is "intentionally ambiguous," in the sense that it can be interpreted as the actual ordered pair consisting of the two numbers x and y, or as *the number* $\langle x, y \rangle = J(x, y)$ that encodes the pair consisting of the two numbers x and y. The context should make it clear which interpretation is intended. In this chapter and the next, it is the number (code) interpretation.

We can define bijections between $\mathbb{N}^n$ and $\mathbb{N}$ by induction for all $n \geq 1$.

Definition 4.2. The function $\langle -, \ldots, - \rangle_n \colon \mathbb{N}^n \to \mathbb{N}$ called an *extended pairing function* is defined as follows. We let

$$\begin{aligned} \langle z \rangle_1 &= z \\ \langle x_1, x_2 \rangle_2 &= \langle x_1, x_2 \rangle, \end{aligned}$$

and

$$\langle x_1, \ldots, x_n, x_{n+1}\rangle_{n+1} = \langle x_1, \ldots, x_{n-1}, \langle x_n, x_{n+1}\rangle\rangle_n,$$

for all $z, x_2, \ldots, x_{n+1} \in \mathbb{N}$.

Again we stress that $\langle x_1, \ldots, x_n\rangle_n$ is a *natural number*. For example.

$$\begin{aligned}
\langle x_1, x_2, x_3\rangle_3 &= \langle x_1, \langle x_2, x_3\rangle\rangle_2 \\
&= \langle x_1, \langle x_2, x_3\rangle\rangle \\
\langle x_1, x_2, x_3, x_4\rangle_4 &= \langle x_1, x_2, \langle x_3, x_4\rangle\rangle_3 \\
&= \langle x_1, \langle x_2, \langle x_3, x_4\rangle\rangle\rangle \\
\langle x_1, x_2, x_3, x_4, x_5\rangle_5 &= \langle x_1, x_2, x_3, \langle x_4, x_5\rangle\rangle_4 \\
&= \langle x_1, \langle x_2, \langle x_3, \langle x_4, x_5\rangle\rangle\rangle\rangle.
\end{aligned}$$

It can be shown by induction on n that

$$\langle x_1, \ldots, x_n, x_{n+1}\rangle_{n+1} = \langle x_1, \langle x_2, \ldots, x_{n+1}\rangle_n\rangle. \tag{$*$}$$

Observe that if $z = \langle x_1, \ldots, x_n\rangle_n$, then $x_1 = \Pi_1(z)$, $x_2 = \Pi_1(\Pi_2(z))$, $x_3 = \Pi_1(\Pi_2(\Pi_2(z)))$, $x_4 = \Pi_1(\Pi_2(\Pi_2(\Pi_2(z))))$, $x_5 = \Pi_2(\Pi_2(\Pi_2(\Pi_2(z))))$.

We can also define a uniform projection function $\Pi\colon \mathbb{N}^3 \to \mathbb{N}$ with the following property: if $z = \langle x_1, \ldots, x_n\rangle$, with $n \geq 2$, then

$$\Pi(i, n, z) = x_i \quad \text{for all } i, \text{ where } 1 \leq i \leq n.$$

The idea is to view z as an n-tuple, and $\Pi(i, n, z)$ as the i-th component of that n-tuple, but if z, n and i do not fit this interpretation, the function must be still be defined and we give it a "crazy" value by default using some simple primitive recursive clauses.

Definition 4.3. The uniform projection function $\Pi\colon \mathbb{N}^3 \to \mathbb{N}$ is defined by cases as follows:

$$\begin{aligned}
&\Pi(i, 0, z) = 0, \quad \text{for all } i \geq 0, \\
&\Pi(i, 1, z) = z, \quad \text{for all } i \geq 0, \\
&\Pi(i, 2, z) = \Pi_1(z), \quad \text{if } 0 \leq i \leq 1, \\
&\Pi(i, 2, z) = \Pi_2(z), \quad \text{for all } i \geq 2,
\end{aligned}$$

and for all $n \geq 2$,

$$\Pi(i, n+1, z) = \begin{cases} \Pi(i, n, z) & \text{if } 0 \leq i < n, \\ \Pi_1(\Pi(n, n, z)) & \text{if } i = n, \\ \Pi_2(\Pi(n, n, z)) & \text{if } i > n. \end{cases}$$

By the results of Section 3.8, this is a legitimate primitive recursive definition. If z is the code $\langle x_1, \ldots, x_{n+1}\rangle_{n+1}$ for the $(n+1)$-tuple $(x_1, \ldots, x_{n+1})$ with $n \geq 2$, then for $0 \leq i < n$, the clause of Definition 4.3 that applies is

$$\Pi(i, n+1, z) = \Pi(i, n, z),$$

and since

$$\langle x_1, \ldots, x_n, x_{n+1}\rangle_{n+1} = \langle x_1, \ldots, x_{n-1}, \langle x_n, x_{n+1}\rangle\rangle_n,$$

we have

$$\begin{aligned}\Pi(i, n+1, \langle x_1, \ldots, x_n, x_{n+1}\rangle_{n+1}) &= \Pi(i, n+1, \langle x_1, \ldots, x_{n-1}, \langle x_n, x_{n+1}\rangle\rangle_n) \\ &= \Pi(i, n, \langle x_1, \ldots, x_{n-1}, \langle x_n, x_{n+1}\rangle\rangle_n),\end{aligned}$$

and since $\langle x_1, \ldots, x_{n-1}, \langle x_n, x_{n+1}\rangle\rangle_n$ codes an n-tuple, for $i = 1, \ldots, n-1$, the value returned is indeed x_i. If $i = n$, then the clause that applies is

$$\Pi(n, n+1, z) = \Pi_1(\Pi(n, n, z)),$$

so we have

$$\begin{aligned}\Pi(n, n+1, \langle x_1, \ldots, x_n, x_{n+1}\rangle_{n+1}) &= \Pi(n, n+1, \langle x_1, \ldots, x_{n-1}, \langle x_n, x_{n+1}\rangle\rangle_n) \\ &= \Pi_1(\Pi(n, n, \langle x_1, \ldots, x_{n-1}, \langle x_n, x_{n+1}\rangle\rangle_n)) \\ &= \Pi_1(\langle x_n, x_{n+1}\rangle) \\ &= x_n.\end{aligned}$$

Finally, if $i = n+1$, then the clause that applies is

$$\Pi(n+1, n+1, z) = \Pi_2(\Pi(n, n, z)),$$

so we have

$$\begin{aligned}\Pi(n+1, n+1, \langle x_1, \ldots, x_n, x_{n+1}\rangle_{n+1}) &= \Pi(n+1, n+1, \langle x_1, \ldots, x_{n-1}, \langle x_n, x_{n+1}\rangle\rangle_n) \\ &= \Pi_2(\Pi(n, n, \langle x_1, \ldots, x_{n-1}, \langle x_n, x_{n+1}\rangle\rangle_n)) \\ &= \Pi_2(\langle x_n, x_{n+1}\rangle) \\ &= x_{n+1}.\end{aligned}$$

When $i = 0$ or $i > n+1$, we get "bogus" values.

Remark: One might argue that it would have been preferable to order the arguments of Π as (n, i, z) rather than (i, n, z). We use the order (i, n, z) in conformity with Machtey and Young [36].

Some basic properties of Π are given as exercises. In particular, the following properties are easily shown:

(a) $\langle 0, \ldots, 0\rangle_n = 0$, $\langle x, 0\rangle = \langle x, 0, \ldots, 0\rangle_n$;

(b) $\Pi(0, n, z) = \Pi(1, n, z)$ and $\Pi(i, n, z) = \Pi(n, n, z)$, for all $i \geq n$ and all $n, z \in \mathbb{N}$;

(c) $\langle \Pi(1, n, z), \ldots, \Pi(n, n, z)\rangle_n = z$, for all $n \geq 1$ and all $z \in \mathbb{N}$;

(d) $\Pi(i, n, z) \leq z$, for all $i, n, z \in \mathbb{N}$;

(e) There is a primitive recursive function Large, such that,

$$\Pi(i, n + 1, \mathrm{Large}(n + 1, z)) = z,$$

for $i, n, z \in \mathbb{N}$.

As a first application, we observe that we need only consider partial computable functions (partial recursive functions)[1] of a single argument. Indeed, let $\varphi\colon \mathbb{N}^n \to \mathbb{N}$ be a partial computable function of $n \geq 2$ arguments. Let $\overline{\varphi}\colon \mathbb{N} \to \mathbb{N}$ be the function given by

$$\overline{\varphi}(z) = \varphi(\Pi(1, n, z), \ldots, \Pi(n, n, z)),$$

for all $z \in \mathbb{N}$. Then $\overline{\varphi}$ is a partial computable function of a single argument, and φ can be recovered from $\overline{\varphi}$, since

$$\varphi(x_1, \ldots, x_n) = \overline{\varphi}(\langle x_1, \ldots, x_n\rangle).$$

Thus, using $\langle -, \cdots, -\rangle$ and Π as coding and decoding functions, we can restrict our attention to functions of a single argument.

Pairing functions can also be used to prove that certain functions are primitive recursive, even though their definition is not a legal primitive recursive definition. For example, consider the *Fibonacci function* defined as follows:

$$\begin{aligned} f(0) &= 1, \\ f(1) &= 1, \\ f(n + 2) &= f(n + 1) + f(n), \end{aligned}$$

for all $n \in \mathbb{N}$. This is not a legal primitive recursive definition, since $f(n + 2)$ depends both on $f(n+1)$ and $f(n)$. In a primitive recursive definition, $g(y+1, \overline{x})$ is only allowed to depend upon $g(y, \overline{x})$, where $\overline{x}$ is an abbrevation for $(x_2, \ldots, x_m)$.

Definition 4.4. Given any function $f\colon \mathbb{N}^n \to \mathbb{N}$, the function $\overline{f}\colon \mathbb{N}^{n+1} \to \mathbb{N}$ defined such that

$$\overline{f}(y, \overline{x}) = \langle f(0, \overline{x}), \ldots, f(y, \overline{x})\rangle_{y+1}$$

is called the *course-of-value function* for f.

The following lemma holds.

[1]The term *partial recursive* is now considered old-fashion. Many researchers have switched to the term *partial computable*.

Proposition 4.1. *Given any function $f\colon \mathbb{N}^n \to \mathbb{N}$, if f is primitive recursive, then so is $\overline{f}$.*

Proof. First it is necessary to define a function *con* such that if $x = \langle x_1, \ldots, x_m \rangle$ and $y = \langle y_1, \ldots, y_n \rangle$, where $m, n \geq 1$, then

$$con(m, x, y) = \langle x_1, \ldots, x_m, y_1, \ldots, y_n \rangle.$$

This fact is left as an exercise. Now, if f is primitive recursive, let

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

showing that $\overline{f}$ is primitive recursive. Conversely, if $\overline{f}$ is primitive recursive, then

$$f(y, \overline{x}) = \Pi(y+1, y+1, \overline{f}(y, \overline{x})),$$

and so, f is primitive recursive. ∎

Remark: Why is it that

$$\overline{f}(y+1, \overline{x}) = \langle \overline{f}(y, \overline{x}), f(y+1, \overline{x}) \rangle$$

does not work? Check the definition of $\langle x_1, \ldots, x_n \rangle_n$.

We define *course-of-value recursion* as follows.

Definition 4.5. Given any two functions $g\colon \mathbb{N}^n \to \mathbb{N}$ and $h\colon \mathbb{N}^{n+2} \to \mathbb{N}$, the function $f\colon \mathbb{N}^{n+1} \to \mathbb{N}$ is defined by *course-of-value recursion* from g and h if

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

The following lemma holds.

Proposition 4.2. *If $f\colon \mathbb{N}^{n+1} \to \mathbb{N}$ is defined by course-of-value recursion from g and h and g, h are primitive recursive, then f is primitive recursive.*

Proof. We prove that $\overline{f}$ is primitive recursive. Then by Proposition 4.1, f is also primitive recursive. To prove that $\overline{f}$ is primitive recursive, observe that

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

When we use Proposition 4.2 to prove that a function is primitive recursive, we rarely bother to construct a formal course-of-value recursion. Instead, we simply indicate how the value of $f(y+1, \overline{x})$ can be obtained in a primitive recursive manner from $f(0, \overline{x})$ through $f(y, \overline{x})$. Thus, an informal use of Proposition 4.2 shows that the Fibonacci function is primitive recursive. A rigorous proof of this fact is left as an exercise.

Next we show that there exist coding and decoding functions between Σ^* and $\{a_1\}^*$, and that partial computable functions over Σ^* can be recoded as partial computable functions over $\{a_1\}^*$. Since $\{a_1\}^*$ is isomorphic to $\mathbb{N}$, this shows that we can restrict out attention to functions defined over $\mathbb{N}$.

4.2 Equivalence of Alphabets

Given an alphabet $\Sigma = \{a_1, \ldots, a_k\}$, strings over Σ can be ordered by viewing strings as numbers in a number system where the digits are $a_1, \ldots, a_k$. In this number system, which is almost the number system with base k, the string a_1 corresponds to zero, and a_k to $k-1$. Hence, we have a kind of shifted number system in base k. The total order on Σ^* induced by this number system is defined so that u precedes v if $|u| < |v|$, and if $|u| = |v|$, then u comes before v in the lexicographic ordering. For example, if $\Sigma = \{a, b, c\}$, a listing of Σ^* in the ordering corresponding to the number system begins with

$$\begin{gathered} a, b, c, aa, ab, ac, ba, bb, bc, ca, cb, cc, \\ aaa, aab, aac, aba, abb, abc, \ldots. \end{gathered}$$

This ordering induces a function from Σ^* to $\mathbb{N}$ which is a bijection. Indeed, if $u = a_{i_1} \cdots a_{i_n}$, this function $f \colon \Sigma^* \to \mathbb{N}$ is given by

$$f(u) = i_1 k^{n-1} + i_2 k^{n-2} + \cdots + i_{n-1} k + i_n.$$

Since we also want a decoding function, we define the coding function $C_k \colon \Sigma^* \to \Sigma^*$ as follows:

$C_k(\epsilon) = \epsilon$, and if $u = a_{i_1} \cdots a_{i_n}$, then

$$C_k(u) = a_1^{i_1 k^{n-1} + i_2 k^{n-2} + \cdots + i_{n-1} k + i_n}.$$

The function C_k is primitive recursive, because

$$\begin{aligned} C_k(\epsilon) &= \epsilon, \\ C_k(x a_i) &= C_k(x)^k a_1^i. \end{aligned}$$

The inverse of C_k is a function $D_k \colon \{a_1\}^* \to \Sigma^*$. However, primitive recursive functions are total, and we need to extend D_k to Σ^*. This is easily done by letting

$$D_k(x) = D_k(a_1^{|x|})$$

for all $x \in \Sigma^*$. It remains to define D_k by primitive recursion over $\Sigma^* = \{a_1, \ldots, a_k\}^*$. For this, we introduce three auxiliary functions p, q, r, defined as follows. Let

$$\begin{aligned} p(\epsilon) &= \epsilon, \\ p(x a_i) &= x a_i, \quad \text{if } i \neq k, \\ p(x a_k) &= p(x). \end{aligned}$$

Note that $p(x)$ is the result of deleting consecutive a_k's in the tail of x. Let

$$\begin{aligned} q(\epsilon) &= \epsilon, \\ q(x a_i) &= q(x) a_1. \end{aligned}$$

Note that $q(x) = a_1^{|x|}$. Finally, let

$$\begin{aligned} r(\epsilon) &= a_1, \\ r(xa_i) &= xa_{i+1}, \quad \text{if } i \neq k, \\ r(xa_k) &= xa_k. \end{aligned}$$

The function r is almost the successor function for the ordering. Then the trick is that $D_k(xa_i)$ is the successor of $D_k(x)$ in the ordering so usually $D_k(xa_i) = r(D_k(x))$, except if

$$D_k(x) = ya_j a_k^n$$

with $j \neq k$, since the successor of $ya_j a_k^n$ is $ya_{j+1}a_1^n$. Thus, we have

$$\begin{aligned} D_k(\epsilon) &= \epsilon, \\ D_k(xa_i) &= r(p(D_k(x)))q(D_k(x) - p(D_k(x))), \quad a_i \in \Sigma. \end{aligned}$$

Then both C_k and D_k are primitive recursive, and $D_k \circ C_k = \text{id}$. Here

$$u - v = \begin{cases} \epsilon & \text{if } |u| \leq |v| \\ w & \text{if } u = xw \text{ and } |x| = |v|. \end{cases}$$

In other words, $u - v$ is u with its first $|v|$ letters deleted. We can show that this function can be defined by primitive recursion by first defining $\text{rdiff}(u, v)$ as v with its first $|u|$ letters deleted, and then

$$u - v = \text{rdiff}(v, u).$$

To define rdiff, we use *tail* given by

$$\begin{aligned} tail(\epsilon) &= \epsilon \\ tail(a_i u) &= u, \quad a_i \in \Sigma,\ u \in \Sigma^*. \end{aligned}$$

We proved in Section 3.7 that *tail* is primitive recursive. Then

$$\begin{aligned} \text{rdiff}(\epsilon, v) &= v \\ \text{rdiff}(ua_i, v) &= \text{rdiff}(u, tail(v)), \quad a_i \in \Sigma. \end{aligned}$$

We leave as an exercise to put all these definitions into the proper format of primitive recursion using projections.

Let $\varphi\colon (\Sigma^*)^n \to \Sigma^*$ be a partial function over Σ^*, and let $\varphi^+\colon (\{a_1\}^*)^n \to \{a_1\}^*$ be the function given by

$$\varphi^+(x_1, \ldots, x_n) = C_k(\varphi(D_k(x_1), \ldots, D_k(x_n))).$$

Also, for any partial function $\psi\colon (\{a_1\}^*)^n \to \{a_1\}^*$, let $\psi^\sharp\colon (\Sigma^*)^n \to \Sigma^*$ be the function given by

$$\psi^\sharp(x_1, \ldots, x_n) = D_k(\psi(C_k(x_1), \ldots, C_k(x_n))).$$

We claim that if ψ is a partial computable function over $(\{a_1\}^*)^n$, then $\psi^\sharp$ is partial computable over $(\Sigma^*)^n$, and that if φ is a partial computable function over $(\Sigma^*)^n$, then φ^+ is partial computable over $(\{a_1\}^*)^n$.

The function ψ can be extended to $(\Sigma^*)^n$ by letting

$$\psi(x_1, \ldots, x_n) = \psi(a_1^{|x_1|}, \ldots, a_1^{|x_n|})$$

for all $x_1, \ldots, x_n \in \Sigma^*$, and so, if ψ is partial computable, then so is the extended function, by composition. It follows that if ψ is partial (or primitive) recursive, then so is $\psi^\sharp$.

This seems equally obvious for φ and φ^+, but there is a difficulty. The problem is that φ^+ is defined as a composition of functions over Σ^*. We have to show how φ^+ can be defined directly over $\{a_1\}^*$ without using any additional alphabet symbols. This is done in Machtey and Young [36], see Section 2.2, Lemma 2.2.3.

4.3 Coding of RAM Programs; The Halting Problem

In this section we present a specific encoding of RAM programs which allows us *to treat programs as integers*. This encoding will allow us to prove one of the most important results of computabilty theory first proven by Turing for Turing machines (1936-1937), the *undecidability of the halting problem* for RAM programs (and Turing machines).

Encoding programs as integers also allows us to have programs that take other programs as input, and we obtain a *universal program*. Universal programs have the property that given two inputs, the first one being the code of a program and the second one an input data, the universal program simulates the actions of the encoded program on the input data. A coding scheme is also called an indexing or a Gödel numbering, in honor to Gödel, who invented this technique.

From results of the previous chapter, without loss of generality, we can restrict out attention to RAM programs computing partial functions of one argument over $\mathbb{N}$. Furthermore, we only need the following kinds of instructions, each instruction being coded as shown below. Since we are considering functions over the natural numbers, which corresponds to a one-letter alphabet, there is only one kind of instruction of the form `add` and `jmp` (`add` increments by 1 the contents of the specified register Rj).

Recall that a conditional jump causes a jump to the closest address Nk above or below iff Rj is nonzero, and if Rj is null, the next instruction is executed. We assume that all lines in a RAM program are numbered. This is always feasible, by labeling unnamed instructions with a new and unused line number.

Definition 4.6. Instructions of a RAM program (operating on $\mathbb{N}$) are coded as follows:

$$
\begin{array}{llcll}
Ni & & \texttt{add} & Rj & \textit{code} = \langle 1, i, j, 0\rangle \\
Ni & & \texttt{tail} & Rj & \textit{code} = \langle 2, i, j, 0\rangle \\
Ni & & \texttt{continue} & & \textit{code} = \langle 3, i, 1, 0\rangle \\
Ni & Rj & \texttt{jmp} & Nka & \textit{code} = \langle 4, i, j, k\rangle \\
Ni & Rj & \texttt{jmp} & Nkb & \textit{code} = \langle 5, i, j, k\rangle
\end{array}
$$

The code of an instruction I is denoted as $\#I$.

To simplify the notation, we introduce the following decoding primitive recursive functions Typ, LNum, Reg, and Jmp, defined as follows:

$$
\begin{aligned}
\mathrm{Typ}(x) &= \Pi(1, 4, x), \\
\mathrm{LNum}(x) &= \Pi(2, 4, x), \\
\mathrm{Reg}(x) &= \Pi(3, 4, x), \\
\mathrm{Jmp}(x) &= \Pi(4, 4, x).
\end{aligned}
$$

The functions yield the type, line number, register name, and line number jumped to, if any, for an instruction coded by x. Note that we have no need to interpret the values of these functions if x does not code an instruction.

We can define the primitive recursive predicate INST, such that $\mathrm{INST}(x)$ holds iff x codes an instruction. First, we need the connective $\Rightarrow$ (*implies*), defined such that

$$P \Rightarrow Q \quad \text{iff} \quad \neg P \vee Q.$$

Definition 4.7. The predicate $\mathrm{INST}(x)$ is defined primitive recursively as follows:

$$
\begin{aligned}
&[1 \leq \mathrm{Typ}(x) \leq 5] \wedge [1 \leq \mathrm{Reg}(x)] \wedge \\
&[\mathrm{Typ}(x) \leq 3 \Rightarrow \mathrm{Jmp}(x) = 0] \wedge \\
&[\mathrm{Typ}(x) = 3 \Rightarrow \mathrm{Reg}(x) = 1].
\end{aligned}
$$

The predicate $\mathrm{INST}(x)$ says that if x is the code of an instruction, say $x = \langle c, i, j, k\rangle$, then $1 \leq c \leq 5$, $j \geq 1$, if $c \leq 3$, then $k = 0$, and if $c = 0$ then we also have $j = 1$.

Definition 4.8. Program are coded as follows. If P is a RAM program composed of the n instructions $I_1, \ldots, I_n$, the code of P, denoted as $\#P$, is

$$\#P = \langle n, \#I_1, \ldots, \#I_n\rangle.$$

Recall from Property $(*)$ in Section 4.1 that

$$\langle n, \#I_1, \ldots, \#I_n\rangle = \langle n, \langle \#I_1, \ldots, \#I_n\rangle\rangle.$$

Also recall that

$$\langle x, y\rangle = ((x + y)^2 + 3x + y)/2.$$

Example 4.1. Consider the following program Padd2 computing the function add2: $\mathbb{N} \to \mathbb{N}$ given by

$$\text{add2}(n) = n + 2.$$

Padd2:

$$\begin{array}{llll} I_1: & 1 & \texttt{add} & R1 \\ I_2: & 2 & \texttt{add} & R1 \\ I_3: & 3 & \texttt{continue} & \end{array}$$

We have

$$\begin{aligned} \#I1 &= \langle 1, 1, 1, 0 \rangle_4 = \langle 1, \langle 1, \langle 1, 0 \rangle \rangle \rangle = 37 \\ \#I2 &= \langle 1, 2, 1, 0 \rangle_4 = \langle 1, \langle 2, \langle 1, 0 \rangle \rangle \rangle = 92 \\ \#I3 &= \langle 3, 3, 1, 0 \rangle_4 = \langle 3, \langle 3, \langle 1, 0 \rangle \rangle \rangle = 234 \end{aligned}$$

and

$$\begin{aligned} \#\text{Padd2} &= \langle 3, \#I1, \#I2, \#I3 \rangle_4 = \langle 3, \langle 37, \langle 92, 234 \rangle \rangle \\ &= 1\,018\,748\,519\,973\,070\,618. \end{aligned}$$

The codes get big fast!

We define the primitive recursive functions Ln, Pg, and Line, such that:

$$\begin{aligned} \text{Ln}(x) &= \Pi(1, 2, x), \\ \text{Pg}(x) &= \Pi(2, 2, x), \\ \text{Line}(i, x) &= \Pi(i, \text{Ln}(x), \text{Pg}(x)). \end{aligned}$$

The function Ln yields the length of the program (the number of instructions), Pg yields the sequence of instructions in the program (really, a code for the sequence), and $\text{Line}(i, x)$ yields the code of the ith instruction in the program. Again, if x does not code a program, there is no need to interpret these functions. However, note that by a previous exercise, it happens that

$$\begin{aligned} \text{Line}(0, x) &= \text{Line}(1, x), \quad \text{and} \\ \text{Line}(\text{Ln}(x), x) &= \text{Line}(i, x), \quad \text{for all } i \geq \text{Ln}(x). \end{aligned}$$

The primitive recursive predicate PROG is defined such that $\text{PROG}(x)$ holds iff x codes a program. Thus, $\text{PROG}(x)$ holds if each line codes an instruction, each jump has an instruction to jump to, and the last instruction is a `continue`.

Definition 4.9. The primitive recursive predicate $\mathrm{PROG}(x)$ is given by

$$\begin{aligned}
&\forall i \leq \mathrm{Ln}(x)[i \geq 1 \Rightarrow \\
&[\mathrm{INST}(\mathrm{Line}(i,x)) \wedge \mathrm{Typ}(\mathrm{Line}(\mathrm{Ln}(x),x)) = 3 \\
&\wedge [\mathrm{Typ}(\mathrm{Line}(i,x)) = 4 \Rightarrow \\
&\exists j \leq i-1[j \geq 1 \wedge \mathrm{LNum}(\mathrm{Line}(j,x)) = \mathrm{Jmp}(\mathrm{Line}(i,x))]]\wedge \\
&[\mathrm{Typ}(\mathrm{Line}(i,x)) = 5 \Rightarrow \\
&\exists j \leq \mathrm{Ln}(x)[j > i \wedge \mathrm{LNum}(\mathrm{Line}(j,x)) = \mathrm{Jmp}(\mathrm{Line}(i,x))]]]].
\end{aligned}$$

Note that we have used Proposition 3.7 which states that if f is a primitive recursive function and if P is a primitive recursive predicate, then $\exists x \leq f(y) P(x)$ is primitive recursive.

The last instruction $\mathrm{Line}(\mathrm{Ln}(x),x))$ in the program must be a `continue`, which means that $\mathrm{Typ}(\mathrm{Line}(\mathrm{Ln}(x),x)) = 3$. When the ith instruction coded by $\mathrm{Line}(i,x)$ of the program coded by x has its first field $\mathrm{Typ}(\mathrm{Line}(i,x)) = 4$, this instruction is a jump above, and there must be an instruction in line j above instruction in line i, which means that $1 \leq j \leq i-1$, and the line number $\mathrm{LNum}(\mathrm{Line}(j,x))$ of the jth instruction must be equal to the jump address $\mathrm{Jmp}(\mathrm{Line}(i,x))$ of the ith instruction. When $\mathrm{Typ}(\mathrm{Line}(i,x)) = 5$, this instruction is a jump below, and the analysis is similar.

We are now ready to prove a fundamental result in the theory of algorithms. This result points out some of the limitations of the notion of algorithm.

Theorem 4.3. *(Undecidability of the halting problem) There is no RAM program* **Decider** *which halts for all inputs and has the following property when started with input x in register $R1$ and with input i in register $R2$ (the other registers being set to zero):*

(1) **Decider** *halts with output* 1 *iff i codes a program that eventually halts when started on input x (all other registers set to zero).*

(2) **Decider** *halts with output* 0 *in $R1$ iff i codes a program that runs forever when started on input x in $R1$ (all other registers set to zero).*

(3) If i does not code a program, then **Decider** *halts with output* 2 *in $R1$.*

Proof. Assume that **Decider** is such a RAM program, and let Q be the following program with a single input:

$$\text{Program } Q \text{ (code } q) \begin{cases} & R2 & \leftarrow & R1 \\ & & P & \\ N1 & & \texttt{continue} & \\ & R1 & \texttt{jmp} & N1a \\ & & \texttt{continue} & \end{cases}$$

Let i be the code of some program P. The key point is that *the termination behavior of Q on input i is exactly the opposite of the termination behavior of* **Decider** *on input i and code i*.

(1) If **Decider** says that program P coded by i *halts* on input i, then $R1$ just after the `continue` in line $N1$ contains 1, and Q *loops forever*.

(2) If **Decider** says that program P coded by i *loops forever* on input i, then $R1$ just after `continue` in line $N1$ contains 0, and Q *halts*.

The program Q can be translated into a program using only instructions of type 1, 2, 3, 4, 5, described previously, and let q *be the code of the program* Q.

Let us see what happens if we run the program Q *on input* q *in* $R1$ *(all other registers set to zero).*

Just after execution of the assignment $R2 \leftarrow R1$, the program **Decider** is started with q *in both* $R1$ and $R2$. Since **Decider** is supposed to halt for all inputs, it eventually halts with output 0 or 1 in $R1$. If **Decider** halts with output 1 in $R1$, then Q goes into an infinite loop, while if **Decider** halts with output 0 in $R1$, then Q halts. But then, because of the definition of **Decider**, we see that **Decider** says that Q halts when started on input q iff Q loops forever on input q, and that Q loops forever on input q iff Q halts on input q, a contradiction. Therefore, **Decider** cannot exist. $\square$

The argument used in the proof of 4.3 is quite similar in spirit to "Russell's Paradox." If we identify the notion of algorithm with that of a RAM program which halts for all inputs, the above theorem says that there is *no* algorithm for deciding whether a RAM program eventually halts for a given input. We say that the halting problem for RAM programs is *undecidable* (or *unsolvable*).

The above theorem also implies that the halting problem for Turing machines is undecidable. Indeed, if we had an algorithm for solving the halting problem for Turing machines, we could solve the halting problem for RAM programs as follows: first, apply the algorithm for translating a RAM program into an equivalent Turing machine, and then apply the algorithm solving the halting problem for Turing machines.

The argument is typical in computability theory and is called a "reducibility argument."

Our next goal is to define a primitive recursive function that describes the computation of RAM programs.

4.4 Universal RAM Programs

To describe the computation of a RAM program, we need to code not only RAM programs but also the contents of the registers. Assume that we have a RAM program P using n registers $R1, \ldots, Rn$, whose contents are denoted as $r_1, \ldots, r_n$. We can code $r_1, \ldots, r_n$ into a single integer $\langle r_1, \ldots, r_n \rangle$. Conversely, every integer x can be viewed as coding the contents of $R1, \ldots, Rn$, by taking the sequence $\Pi(1, n, x), \ldots, \Pi(n, n, x)$.

Actually, it is not necessary to know n, the number of registers, if we make the following observation:

$$\mathrm{Reg}(\mathrm{Line}(i,x)) \le \mathrm{Line}(i,x) \le \mathrm{Pg}(x) < x$$

for all $i, x \in \mathbb{N}$. If x codes a program, then $R1, \ldots, Rx$ certainly include all the registers in the program. Also note that from a previous exercise,

$$\langle r_1, \ldots, r_n, 0, \ldots, 0\rangle = \langle r_1, \ldots, r_n, 0\rangle.$$

We now define the primitive recursive functions Nextline, Nextcont, and Comp, describing the computation of RAM programs. There are a lot of tedious technical details that the reader should skip upon first reading. However, to be rigorous, we must spell out all these details.

Definition 4.10. Let x code a program and let i be such that $1 \le i \le \mathrm{Ln}(x)$. The following functions are defined:

(1) $\mathrm{Nextline}(i,x,y)$ is the number of the next instruction to be executed after executing the ith instruction (the current instruction) in the program coded by x, where the contents of the registers is coded by y.

(2) $\mathrm{Nextcont}(i,x,y)$ is the code of the contents of the registers after executing the ith instruction in the program coded by x, where the contents of the registers is coded by y.

(3) $\mathrm{Comp}(x,y,m) = \langle i, z\rangle$, where i and z are defined such that after running the program coded by x for m steps, where the initial contents of the program registers are coded by y, the next instruction to be executed is the ith one, and z is the code of the current contents of the registers.

Proposition 4.4. *The functions* Nextline*,* Nextcont*, and* Comp *are primitive recursive.*

Proof. (1) $\mathrm{Nextline}(i,x,y) = i+1$, unless the ith instruction is a jump and the contents of the register being tested is nonzero:

$$\begin{aligned}
&\mathrm{Nextline}(i,x,y) = \\
&\qquad\qquad \max j \le \mathrm{Ln}(x)[j < i \land \mathrm{LNum}(\mathrm{Line}(j,x)) = \mathrm{Jmp}(\mathrm{Line}(i,x))] \\
&\qquad\qquad \text{if } \mathrm{Typ}(\mathrm{Line}(i,x)) = 4 \land \Pi(\mathrm{Reg}(\mathrm{Line}(i,x)), x, y) \neq 0 \\
&\qquad\qquad \min j \le \mathrm{Ln}(x)[j > i \land \mathrm{LNum}(\mathrm{Line}(j,x)) = \mathrm{Jmp}(\mathrm{Line}(i,x))] \\
&\qquad\qquad \text{if } \mathrm{Typ}(\mathrm{Line}(i,x)) = 5 \land \Pi(\mathrm{Reg}(\mathrm{Line}(i,x)), x, y) \neq 0 \\
&\qquad\qquad i+1 \text{ otherwise.}
\end{aligned}$$

For example, if the ith instruction of the program coded by x is a jump above, namely $\mathrm{Typ}(\mathrm{Line}(i,x)) = 4$, then the register being tested is $\mathrm{Reg}(\mathrm{Line}(i,x))$, and its contents must be nonzero for a jump to occur, so the contents of this register, which is obtained from the code y of all registers as $\Pi(\mathrm{Reg}(\mathrm{Line}(i,x)), x, y)$ (remember that we may assume that there are x registers, by padding with zeros) must be nonzero.

Note that according to this definition, if the ith line is the final `continue`, then Nextline signals that the program has halted by yielding

$$\text{Nextline}(i, x, y) > \text{Ln}(x).$$

(2) We need two auxiliary functions Add and Sub defined as follows.

$\text{Add}(j, x, y)$ is the number coding the contents of the registers used by the program coded by x after register Rj coded by $\Pi(j, x, y)$ has been increased by 1, and

$\text{Sub}(j, x, y)$ codes the contents of the registers after register Rj has been decremented by 1 (y codes the previous contents of the registers). It is easy to see that

$$\begin{aligned}\text{Sub}(j, x, y) = \min z \leq y[&\Pi(j, x, z) = \Pi(j, x, y) - 1\\ &\wedge \forall k \leq x[0 < k \neq j \Rightarrow \Pi(k, x, z) = \Pi(k, x, y)]].\end{aligned}$$

The definition of Add is slightly more tricky. We leave as an exercise to the reader to prove that:

$$\begin{aligned}\text{Add}(j, x, y) = {}&\min z \leq \text{Large}(x, y + 1)\\ &[\Pi(j, x, z) = \Pi(j, x, y) + 1 \wedge \forall k \leq x[0 < k \neq j \Rightarrow \Pi(k, x, z) = \Pi(k, x, y)]],\end{aligned}$$

where the function Large is the function defined in an earlier exercise. Then

$$\begin{aligned}&\text{Nextcont}(i, x, y) =\\ &\qquad \begin{array}{l}\text{Add}(\text{Reg}(\text{Line}(i, x), x, y) \quad \text{if} \quad \text{Typ}(\text{Line}(i, x)) = 1\\ \text{Sub}(\text{Reg}(\text{Line}(i, x), x, y) \quad \text{if} \quad \text{Typ}(\text{Line}(i, x)) = 2\\ y \quad \text{if} \quad \text{Typ}(\text{Line}(i, x)) \geq 3.\end{array}\end{aligned}$$

(3) Recall that $\Pi_1(z) = \Pi(1, 2, z)$ and $\Pi_2(z) = \Pi(2, 2, z)$. The function Comp is defined by primitive recursion as follows:

$$\begin{aligned}\text{Comp}(x, y, 0) &= \langle 1, y\rangle\\ \text{Comp}(x, y, m + 1) &= \langle \text{Nextline}(\Pi_1(\text{Comp}(x, y, m)), x, \Pi_2(\text{Comp}(x, y, m))),\\ &\qquad \text{Nextcont}(\Pi_1(\text{Comp}(x, y, m)), x, \Pi_2(\text{Comp}(x, y, m)))\rangle.\end{aligned}$$

If $\text{Comp}(x, y, m) = \langle i, z\rangle$, then $\Pi_1(\text{Comp}(x, y, m)) = i$ is the number of the next instruction to be executed and $\Pi_2(\text{Comp}(x, y, m)) = z$ codes the current contents of the registers, so

$$\text{Comp}(x, y, m + 1) = \langle \text{Nextline}(i, x, z), \text{Nextcont}(i, x, z)\rangle,$$

as desired. $\square$

We can now reprove that every RAM computable function is partial computable. Indeed, assume that x codes a program P.

We would like to define the partial function End so that for all x, y, where x codes a program and y codes the contents of its registers, $\mathrm{End}(x, y)$ is the number of steps for which the computation runs before halting, if it halts. If the program does not halt, then $\mathrm{End}(x, y)$ is undefined.

If y is the value of the register $R1$ before the program P coded by x is started, recall that the contents of the registers is coded by $\langle y, 0\rangle$. Noticing that 0 and 1 do not code programs, we note that if x codes a program, then $x \geq 2$, and $\Pi_1(z) = \Pi(1, x, z)$ is the contents of $R1$ as coded by z.

Since $\mathrm{Comp}(x, y, m) = \langle i, z\rangle$, we have

$$\Pi_1(\mathrm{Comp}(x, y, m)) = i,$$

where i is the number (index) of the instruction reached after running the program P coded by x with initial values of the registers coded by y for m steps. Thus, P halts if i is the last instruction in P, namely $\mathrm{Ln}(x)$, iff

$$\Pi_1(\mathrm{Comp}(x, y, m)) = \mathrm{Ln}(x).$$

This suggests the following definition.

Definition 4.11. The predicate $\mathrm{End}(x, y)$ is defined by

$$\mathrm{End}(x, y) = \min m[\Pi_1(\mathrm{Comp}(x, y, m)) = \mathrm{Ln}(x)].$$

Note that End is a partial computable function; it can be computed by a RAM program involving *only one while loop* searching for the number of steps m. The function involved in the minimization is *primitive recursive*. However, in general, End is not a total function.

If φ is the partial computable function computed by the program P coded by x, then we claim that

$$\varphi(y) = \Pi_1(\Pi_2(\mathrm{Comp}(x, \langle y, 0\rangle, \mathrm{End}(x, \langle y, 0\rangle)))).$$

This is because if $m = \mathrm{End}(x, \langle y, 0\rangle)$ is the number of steps after which the program P coded by x halts on input y, then

$$\mathrm{Comp}(x, \langle y, 0\rangle, m)) = \langle \mathrm{Ln}(x), z\rangle,$$

where z is the code of the register contents when the program stops. Consequently

$$\begin{aligned} z &= \Pi_2(\mathrm{Comp}(x, \langle y, 0\rangle, m)) \\ z &= \Pi_2(\mathrm{Comp}(x, \langle y, 0\rangle, \mathrm{End}(x, \langle y, 0\rangle))). \end{aligned}$$

The value of the register $R1$ is $\Pi_1(z)$, that is

$$\varphi(y) = \Pi_1(\Pi_2(\mathrm{Comp}(x, \langle y, 0\rangle, \mathrm{End}(x, \langle y, 0\rangle)))).$$

The above fact is worth recording as the following proposition which is a variant of a result known as the *Kleene normal form*

Proposition 4.5. *(Kleene normal form for RAM programs) If φ is the partial computable function computed by the program P coded by x, then we have*

$$\varphi(y) = \Pi_1(\Pi_2(\mathrm{Comp}(x, \langle y, 0\rangle, \mathrm{End}(x, \langle y, 0\rangle))) \quad \textit{for all } y \in \mathbb{N}.$$

Observe that φ is written in the form $\varphi = g \circ \min f$, for some primitive recursive functions f and g. It will be convenient to denote the function φ computed by the RAM program coded by x as φ_x.

We can also exhibit a partial computable function which enumerates all the unary partial computable functions. It is a *universal function*.

Abusing the notation slightly, we will write $\varphi(x, y)$ for $\varphi(\langle x, y\rangle)$, viewing φ as a function of two arguments (however, φ is really a function of a single argument). We define the function φ_{univ} as follows:

$$\varphi_{univ}(x, y) = \begin{cases} \Pi_1(\Pi_2(\mathrm{Comp}(x, \langle y, 0\rangle, \mathrm{End}(x, \langle y, 0\rangle))) & \text{if PROG}(x), \\ \text{undefined} & \text{otherwise.} \end{cases}$$

The function φ_{univ} is a partial computable function with the following property: for every x coding a RAM program P, for every input y,

$$\varphi_{univ}(x, y) = \varphi_x(y),$$

the value of the partial computable function φ_x computed by the RAM program P coded by x. If x does not code a program, then $\varphi_{univ}(x, y)$ is undefined for all y.

By Proposition 3.9, the partial function φ_{univ} is not computable (recursive).[2] Indeed, being an enumerating function for the partial computable functions, it is an enumerating function for the total computable functions, and thus, it cannot be computable. Being a partial function saves us from a contradiction.

The existence of the universal function φ_{univ} is sufficiently important to be recorded in the following proposition.

Proposition 4.6. *(Universal RAM program) For the indexing of RAM programs defined earlier, there is a universal partial computable function φ_{univ} such that, for all $x, y \in \mathbb{N}$, if φ_x is the partial computable function computed by P_x, then*

$$\varphi_x(y) = \varphi_{univ}(\langle x, y\rangle).$$

The program UNIV computing φ_{univ} can be viewed as an *interpreter* for RAM programs. By giving the universal program UNIV the "program" x and the "data" y, we get the result of executing program P_x on input y. We can view the RAM model as a *stored program computer*.

[2]The term *recursive function* is now considered old-fashion. Many researchers have switched to the term *computable function*.

By Theorem 4.3 and Proposition 4.6, the halting problem for the single program UNIV is undecidable. Otherwise, the halting problem for RAM programs would be decidable, a contradiction. It should be noted that the program UNIV can actually be written (with a certain amount of pain).

The existence of the function φ_{univ} leads us to the notion of an *indexing* of the RAM programs.

4.5 Indexing of RAM Programs

We can define a listing of the RAM programs as follows. If x codes a program (that is, if PROG(x) holds) and P is the program that x codes, we call this program P the xth RAM program and denote it as P_x. If x does not code a program, we let P_x be the program that diverges for every input:

$$\begin{array}{llcl} N1 & & \texttt{add} & R1 \\ N1 & R1 & \texttt{jmp} & N1a \\ N1 & & \texttt{continue} & \end{array}$$

Therefore, in all cases, P_x stands for the xth RAM program. Thus, we have a listing of RAM programs, $P_0, P_1, P_2, P_3, \ldots$, such that every RAM program (of the restricted type considered here) appears in the list exactly once, except for the "infinite loop" program. For example, the program Padd2 (adding 2 to an integer) appears as

$$P_{1\,018\,748\,519\,973\,070\,618}.$$

In particular, note that φ_{univ} being a partial computable function, it is computed by some RAM program UNIV that has a code *univ* and is the program P_{univ} in the list.

Having an indexing of the RAM programs, we also have an indexing of the partial computable functions.

Definition 4.12. For every integer $x \geq 0$, we let P_x be the RAM program coded by x as defined earlier, and φ_x be the partial computable function computed by P_x.

For example, the function add2 (adding 2 to an integer) appears as

$$\varphi_{1\,018\,748\,519\,973\,070\,618}.$$

Remark: Kleene used the notation $\{x\}$ for the partial computable function coded by x. Due to the potential confusion with singleton sets, we follow Rogers, and use the notation φ_x; see Rogers [44], page 21.

It is important to observe that *different programs* P_x and P_y may compute the *same function*, that is, while $P_x \neq P_y$ for all $x \neq y$, it is possible that $\varphi_x = \varphi_y$. For example, the program P_y coded by y may be the program obtained from the program P_x coded by x obtained by adding and subtracting 1 a million times to a register not in the program P_x. In fact, it is *undecidable* whether $\varphi_x = \varphi_y$.

The object of the next section is to show the existence of Kleene's T-predicate. This will yield another important normal form. In addition, the T-predicate is a basic tool in recursion theory.

4.6 Kleene's T-Predicate

In Section 4.3, we have encoded programs. The idea of this section is to also encode *computations* of RAM programs. Assume that x codes a program, that y is some input (not a code), and that z codes a computation of P_x on input y.

Definition 4.13. The predicate $T(x, y, z)$ is defined as follows:

$T(x, y, z)$ holds iff x codes a RAM program, y is an input, and z codes a halting computation of P_x on input y.

The code z of a computation packs the consecutive "states" of the computation, namely the pairs $\langle i_j, y_j \rangle$, where i_j is the physical location of the next instruction to be executed and each y_j codes the contents of the registers just before execution of this instruction. We will show that T is primitive recursive.

First we need to *encode computations*. We say that z codes a computation of length $n \geq 1$ if

$$z = \langle n + 2, \langle 1, y_0 \rangle, \langle i_1, y_1 \rangle, \ldots, \langle i_n, y_n \rangle \rangle,$$

where each i_j is the physical location of the next instruction to be executed and each y_j codes the contents of the registers just before execution of the instruction at the location i_j. Also, y_0 codes the initial contents of the registers, that is, $y_0 = \langle y, 0 \rangle$, for some input y.

We let $\mathrm{Lz}(z) = \Pi_1(z)$ (not to be confused with $\mathrm{Ln}(x)$).

Note that i_j denotes the physical location of the next instruction to be executed in the sequence of instructions constituting the program coded by x, and not the line number (label) of this instruction. Thus, the first instruction to be executed is in location 1, $1 \leq i_j \leq \mathrm{Ln}(x)$, and $i_{n-1} = \mathrm{Ln}(x)$. Since the last instruction which is executed is the last physical instruction in the program, namely, a `continue`, there is no next instruction to be executed after that, and i_n is irrelevant. Writing the definition of T is a little simpler if we let $i_n = \mathrm{Ln}(x) + 1$.

Definition 4.14. The T-predicate is the primitive recursive predicate defined as follows:

$T(x,y,z)$ iff PROG(x) and $(\mathrm{Lz}(z) \geq 3)$ and
$\forall j \leq \mathrm{Lz}(z) - 3[0 \leq j \Rightarrow$
$\mathrm{Nextline}(\Pi_1(\Pi(j+2, \mathrm{Lz}(z), z)), x, \Pi_2(\Pi(j+2, \mathrm{Lz}(z), z))) = \Pi_1(\Pi(j+3, \mathrm{Lz}(z), z))$ and
$\mathrm{Nextcont}(\Pi_1(\Pi(j+2, \mathrm{Lz}(z), z)), x, \Pi_2(\Pi(j+2, \mathrm{Lz}(z), z))) = \Pi_2(\Pi(j+3, \mathrm{Lz}(z), z))$ and
$\Pi_1(\Pi(\mathrm{Lz}(z)-1, \mathrm{Lz}(z), z)) = \mathrm{Ln}(x)$ and
$\Pi_1(\Pi(2, \mathrm{Lz}(z), z)) = 1$ and
$y = \Pi_1(\Pi_2(\Pi(2, \mathrm{Lz}(z), z)))$ and $\Pi_2(\Pi_2(\Pi(2, \mathrm{Lz}(z), z))) = 0]$.

The reader can verify that $T(x,y,z)$ holds iff x codes a RAM program, y is an input, and z codes a halting computation of P_x on input y. For example, since

$$z = \langle n+2, \langle 1, y_0\rangle, \langle i_1, y_1\rangle, \ldots, \langle i_n, y_n\rangle\rangle,$$

we have $\Pi(j+2, \mathrm{Lz}(z), z) = \langle i_{j-1}, y_{j-1}\rangle$ and $\Pi(j+3, \mathrm{Lz}(z), z) = \langle i_j, y_j\rangle$, so $\Pi_1(\Pi(j+2, \mathrm{Lz}(z), z)) = \Pi_1(\langle i_{j-1}, y_{j-1}\rangle) = i_{j-1}$, $\Pi_2(\Pi(j+2, \mathrm{Lz}(z), z)) = \Pi_2(\langle i_{j-1}, y_{j-1}\rangle) = y_{j-1}$, and similarly $\Pi_1(\Pi(j+3, \mathrm{Lz}(z), z)) = i_j$, $\Pi_2(\Pi(j+3, \mathrm{Lz}(z), z)) = y_j$, so the T predicate expresses that $\mathrm{Nextline}(i_{j-1}, y_{j-1}) = i_j$ and $\mathrm{Nextcont}(i_{j-1}, y_{j-1}) = y_j$.

In order to extract the output of P_x from z, we define the primitive recursive function Res as follows:

$$\mathrm{Res}(z) = \Pi_1(\Pi_2(\Pi(\mathrm{Lz}(z), \mathrm{Lz}(z), z))).$$

The explanation for this formula is that if $\Pi(\mathrm{Lz}(z), \mathrm{Lz}(z), z) = \langle i_n, y_n\rangle$, then $\Pi_2(\Pi(\mathrm{Lz}(z), \mathrm{Lz}(z), z)) = y_n$, the code of the registers, and since the output is returned in Register $R1$, $\mathrm{Res}(z)$ is the contents of register $R1$ when P_x halts, that is, $\Pi_1(y_{\mathrm{Lz}(z)})$. Using the T-predicate, we get the so-called Kleene normal form.

Theorem 4.7. *(Kleene Normal Form) Using the indexing of the partial computable functions defined earlier, we have*

$$\varphi_x(y) = \mathrm{Res}[\min z(T(x,y,z))],$$

where $T(x,y,z)$ and Res *are primitive recursive.*

Note that the universal function φ_{univ} can be defined as

$$\varphi_{univ}(x,y) = \mathrm{Res}[\min z(T(x,y,z))].$$

There is another important property of the partial computable functions, namely, that composition is effective (computable). We need two auxiliary primitive recursive functions. The function Conprogs creates the code of the program obtained by concatenating the programs P_x and P_y, and for $i \geq 2$, $\mathrm{Cumclr}(i)$ is the code of the program which clears registers $R2, \ldots, Ri$. To get Cumclr, we can use the function $\mathrm{clr}(i)$ such that $\mathrm{clr}(i)$ is the code of the program

$N1$		`tail`	Ri
$N1$	Ri	`jmp`	$N1a$
N		`continue`	

We leave it as an exercise to prove that clr, Conprogs, and Cumclr, are primitive recursive.

Theorem 4.8. *There is a primitive recursive function c such that*

$$\varphi_{c(x,y)} = \varphi_x \circ \varphi_y.$$

Proof. If both x and y code programs, then $\varphi_x \circ \varphi_y$ can be computed as follows: Run P_y, clear all registers but $R1$, then run P_x. Otherwise, let loop be the index of the infinite loop program:

$$c(x,y) = \begin{cases} \mathrm{Conprogs}(y, \mathrm{Conprogs}(\mathrm{Cumclr}(y), x)) & \text{if } \mathrm{PROG}(x) \text{ and } \mathrm{PROG}(y) \\ \mathrm{loop} & \text{otherwise.} \end{cases}$$

∎

4.7 A Non-Computable Function; Busy Beavers

Total functions that *are not computable must grow very fast and thus are very complicated*. Yet, in 1962, Radó published a paper in which he defined two functions Σ and S (involving computations of Turing machines) that are total and not computable.

Consider Turing machines with a tape alphabet $\Gamma = \{1, B\}$ with two symbols (B being the blank). We also assume that these Turing machines have a special final state q_F, which is a blocking state (there are no transitions from q_F). We do not count this state when counting the number of states of such Turing machines. The game is to run such Turing machines with a fixed number of states n starting on a blank tape, with the goal of producing the maximum number of (not necessarily consecutive) ones (1).

Definition 4.15. The function Σ (defined on the positive natural numbers) is defined as the maximum number $\Sigma(n)$ of (not necessarily consecutive) 1's written on the tape after a Turing machine with $n \geq 1$ states started on the blank tape halts. The function S is defined as the maximum number $S(n)$ of moves that can be made by a Turing machine of the above type with n states before it halts, started on the blank tape.[3]

Definition 4.16. A Turing machine with n states that writes the maximum number $\Sigma(n)$ of 1's when started on the blank tape is called a *busy beaver*.

[3]The function S defined here is obviously not the successor function from Definition 3.13.

Busy beavers are hard to find, even for small n. First, it can be shown that the number of distinct Turing machines of the above kind with n states is $(4(n+1))^{2n}$. Second, since it is undecidable whether a Turing machine halts on a given input, it is hard to tell which machines loop or halt after a very long time.

Here is a summary of what is known for $1 \leq n \leq 6$. Observe that the exact value of $\Sigma(5), \Sigma(6), S(5)$ and $S(6)$ is unknown.

n	$\Sigma(n)$	$S(n)$
1	1	1
2	4	6
3	6	21
4	13	107
5	≥ 4098	$\geq 47,176,870$
6	$\geq 95,524,079$	$\geq 8,690,333,381,690,951$
6	$\geq 3.515 \times 10^{18267}$	$\geq 7.412 \times 10^{36534}$

The first entry in the table for $n = 6$ corresponds to a machine due to Heiner Marxen (1999). This record was surpassed by Pavel Kropitz in 2010, which corresponds to the second entry for $n = 6$. The machines achieving the record in 2017 for $n = 4, 5, 6$ are shown below, where the blank is denoted Δ instead of B, and where the special halting state is denoted H:

4-state busy beaver:

	A	B	C	D
Δ	$(1, R, B)$	$(1, L, A)$	$(1, R, H)$	$(1, R, D)$
1	$(1, L, B)$	(Δ, L, C)	$(1, L, D)$	(Δ, R, A)

The above machine output 13 ones in 107 steps. In fact, the output is

$$\Delta\,\Delta\,1\,\Delta\,1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,\Delta\,\Delta.$$

5-state best contender:

	A	B	C	D	E
Δ	$(1, R, B)$	$(1, R, C)$	$(1, R, D)$	$(1, L, A)$	$(1, R, H)$
1	$(1, L, C)$	$(1, R, B)$	(Δ, L, E)	$(1, L, D)$	(Δ, L, A)

The above machine output 4098 ones in $47, 176, 870$ steps.

6-state contender (Heiner Marxen):

	A	B	C	D	E	F
Δ	$(1, R, B)$	$(1, L, C)$	(Δ, R, F)	$(1, R, A)$	$(1, L, H)$	(Δ, L, A)
1	$(1, R, A)$	$(1, L, B)$	$(1, L, D)$	(Δ, L, E)	$(1, L, F)$	(Δ, L, C)

The above machine outputs 96, 524, 079 ones in 8, 690, 333, 381, 690, 951 steps.

6-state best contender (Pavel Kropitz):

	A	B	C	D	E	F
Δ	$(1, R, B)$	$(1, R, C)$	$(1, L, D)$	$(1, R, E)$	$(1, L, A)$	$(1, L, H)$
1	$(1, L, E)$	$(1, R, F)$	(Δ, R, B)	(Δ, L, C)	(Δ, R, D)	$(1, R, C)$

The above machine output at least 3.515×10^{18267} ones!

The reason why it is so hard to compute Σ and S is that they are not computable!

Theorem 4.9. *The functions Σ and S are total functions that are not computable (not recursive).*

Proof sketch. The proof consists in showing that Σ (and similarly for S) eventually outgrows any computable function. More specifically, we claim that for every computable function f, there is some positive integer k_f such that

$$\Sigma(n + k_f) \geq f(n) \quad \text{for all } n \geq 0.$$

We simply have to pick k_f to be the number of states of a Turing machine M_f computing f. Then we can create a Turing machine $M_{n,f}$ that works as follows. Using n of its states, it writes n ones on the tape, and then it simulates M_f with input 1^n. Since the ouput of $M_{n,f}$ started on the blank tape consists of $f(n)$ ones, and since $\Sigma(n + k_f)$ is the maximum number of ones that a turing machine with $n + k_f$ states will ouput when it stops, we must have

$$\Sigma(n + k_f) \geq f(n) \quad \text{for all } n \geq 0.$$

Next observe that $\Sigma(n) < \Sigma(n + 1)$, because we can create a Turing machine with $n + 1$ states which simulates a busy beaver machine with n states, and then writes an extra 1 when the busy beaver stops, by making a transition to the $(n+1)$th state. It follows immediately that if $m < n$ then $\Sigma(m) < \Sigma(n)$. If Σ was computable, then so would be the function g given by $g(n) = \Sigma(2n)$. By the above, we would have

$$\Sigma(n + k_g) \geq g(n) = \Sigma(2n) \quad \text{for all } n \geq 0,$$

and for $n > k_g$, since $2n > n + k_g$, we would have $\Sigma(n + n_g) < \Sigma(2n)$, contradicting the fact that $\Sigma(n + n_g) \geq \Sigma(2n)$.

Since by definition $S(n)$ is the maximum number of moves that can be made by a Turing machine of the above type with n states before it halts, $S(n) \geq \Sigma(n)$. Then the same reasoning as above shows that S is not a computable function. $\square$

The zoo of computable and non-computable functions is illustrated in Figure 4.1.

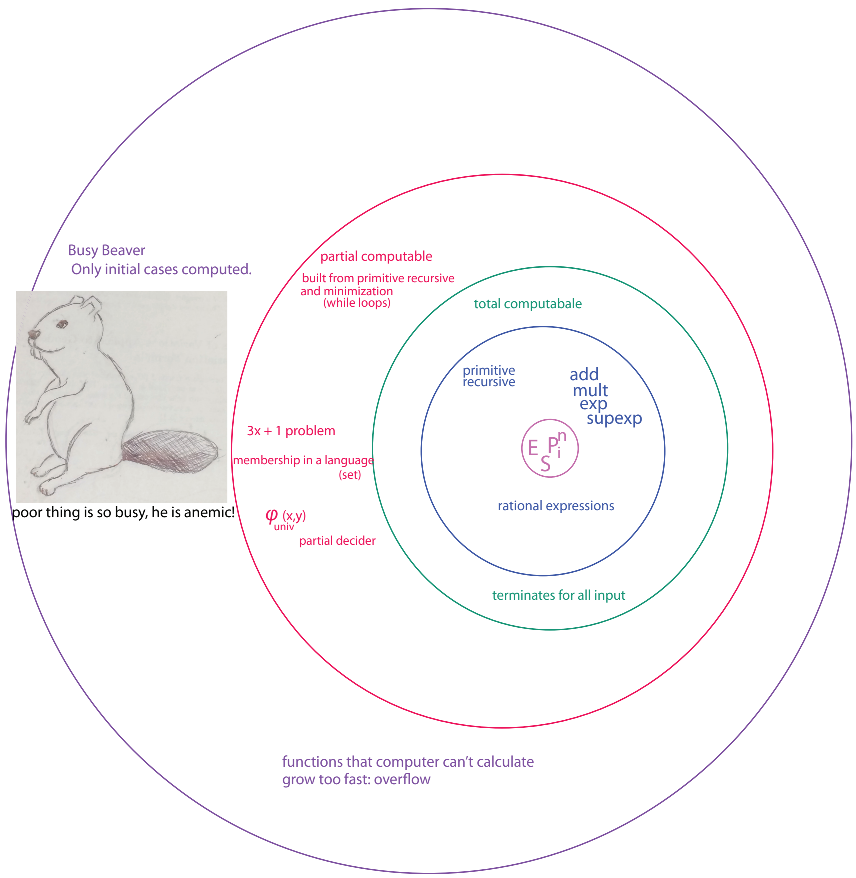


Figure 4.1: Computability Classification of Functions.

Chapter 5

Elementary Recursive Function Theory

5.1 Acceptable Indexings

In Chapter 4, we have exhibited a specific indexing of the partial computable functions by encoding the RAM programs. Using this indexing, we showed the existence of a universal function φ_{univ} and of a computable function c, with the property that for all $x, y \in \mathbb{N}$,

$$\varphi_{c(x,y)} = \varphi_x \circ \varphi_y.$$

It is natural to wonder whether the same results hold if a different coding scheme is used or if a different model of computation is used, for example, Turing machines. In other words, we would like to know if our results depend on a specific coding scheme or not.

Our previous results showing the characterization of the partial computable functions being independent of the specific model used, suggests that it might be possible to pinpoint certain properties of coding schemes which would allow an axiomatic development of recursive function theory. What we are aiming at is to find some simple properties of "nice" coding schemes that allow one to proceed without using explicit coding schemes, as long as the above properties hold.

Remarkably, such properties exist. Furthermore, any two coding schemes having these properties are equivalent in a strong sense (called effectively equivalent), and so, one can pick any such coding scheme without any risk of losing anything else because the wrong coding scheme was chosen. Such coding schemes, also called indexings, or Gödel numberings, or even programming systems, are called *acceptable indexings*.

Definition 5.1. An *indexing* of the partial computable functions is an infinite sequence $\varphi_0, \varphi_1, \ldots,$ of partial computable functions that includes *all* the partial computable functions of one argument (there might be repetitions, this is why we are not using the term enumeration). An indexing is *universal* if it contains the partial computable function φ_{univ}

such that

$$\varphi_{univ}(i, x) = \varphi_i(x) \quad \text{for all } i, x \in \mathbb{N}. \tag{$*_{\rm univ}$}$$

An indexing is *acceptable* if it is universal and if there is a total computable function c for composition, such that

$$\varphi_{c(i,j)} = \varphi_i \circ \varphi_j \quad \text{for all } i, j \in \mathbb{N}. \tag{$*_{\rm compos}$}$$

An indexing may fail to be universal because it is not "computable enough," in the sense that it does not yield a function φ_{univ} satisfying ($*_{\rm univ}$). It may also fail to be acceptable because it is not "computable enough," in the sense that it does not yield a function φ_{univ} satisfying ($*_{\rm compos}$).

From Chapter 4, we know that the specific indexing of the partial computable functions given for RAM programs is acceptable. Another characterization of acceptable indexings left as an exercise is the following: an indexing $\psi_0, \psi_1, \psi_2, \ldots$ of the partial computable functions is acceptable iff there exists a total computable function f translating the RAM indexing of Section 4.3 into the indexing $\psi_0, \psi_1, \psi_2, \ldots$, that is,

$$\varphi_i = \psi_{f(i)} \quad \text{for all } i \in \mathbb{N}.$$

A very useful property of acceptable indexings is the so-called "s-m-n Theorem". Using the slightly loose notation $\varphi(x_1, \ldots, x_n)$ for $\varphi(\langle x_1, \ldots, x_n\rangle)$, the s-m-n Theorem says the following. Given a function φ considered as having $m + n$ arguments, if we fix the values of the first m arguments and we let the other n arguments vary, we obtain a function ψ of n arguments. Then the index of ψ depends in a computable fashion upon the index of φ and the first m arguments $x_1, \ldots, x_m$. We can "pull" the first m arguments of φ into the index of ψ.

Theorem 5.1. *(The "s-m-n Theorem") For any acceptable indexing $\varphi_0, \varphi_1, \ldots$, there is a total computable function $s\colon \mathbb{N}^{n+2} \to \mathbb{N}$, such that, for all $i, m, n \geq 1$, for all $x_1, \ldots, x_m$ and all $y_1, \ldots, y_n$, we have*

$$\varphi_{s(i,m,x_1,\ldots,x_m)}(y_1, \ldots, y_n) = \varphi_i(x_1, \ldots, x_m, y_1, \ldots, y_n).$$

Proof. First, note that the above identity is really

$$\varphi_{s(i,m,\langle x_1,\ldots,x_m\rangle)}(\langle y_1, \ldots, y_n\rangle) = \varphi_i(\langle x_1, \ldots, x_m, y_1, \ldots, y_n\rangle).$$

Recall that there is a primitive recursive function Con such that

$$\mathrm{Con}(m, \langle x_1, \ldots, x_m\rangle, \langle y_1, \ldots, y_n\rangle) = \langle x_1, \ldots, x_m, y_1, \ldots, y_n\rangle$$

for all $x_1, \ldots, x_m, y_1, \ldots, y_n \in \mathbb{N}$. Hence, a computable function s such that

$$\varphi_{s(i,m,x)}(y) = \varphi_i(\mathrm{Con}(m, x, y))$$

will do. We define some auxiliary primitive recursive functions as follows:

$$P(y) = \langle 0, y\rangle \quad \text{and} \quad Q(\langle x, y\rangle) = \langle x+1, y\rangle.$$

Since we have an indexing of the partial computable functions, there are indices p and q such that $P = \varphi_p$ and $Q = \varphi_q$. Let R be defined such that

$$\begin{aligned} R(0) &= p, \\ R(x+1) &= c(q, R(x)), \end{aligned}$$

where c is the computable function for composition given by the indexing. We prove by induction of x that

$$\varphi_{R(x)}(y) = \langle x, y\rangle \quad \text{for all } x, y \in \mathbb{N}.$$

For this we use the existence of the universal function φ_{univ}.

For the base case $x = 0$, we have

$$\begin{aligned} \varphi_{R(0)}(y) &= \varphi_{univ}(\langle R(0), y\rangle) \\ &= \varphi_{univ}(\langle p, y\rangle) \\ &= \varphi_p(y) = P(y) = \langle 0, y\rangle. \end{aligned}$$

For the induction step, we have

$$\begin{aligned} \varphi_{R(x+1)}(y) &= \varphi_{univ}(\langle R(x+1), y\rangle) \\ &= \varphi_{univ}(\langle c(q, R(x)), y\rangle) \\ &= \varphi_{c(q,R(x))}(y) \\ &= (\varphi_q \circ \varphi_{R(x)})(y) \\ &= \varphi_q(\langle x, y\rangle) = Q(\langle x, y\rangle) = \langle x+1, y\rangle. \end{aligned}$$

Also, recall that $\langle x, y, z\rangle = \langle x, \langle y, z\rangle\rangle$, by definition of pairing. Then we have

$$\varphi_{R(x)} \circ \varphi_{R(y)}(z) = \varphi_{R(x)}(\langle y, z\rangle) = \langle x, y, z\rangle.$$

Finally, let k be an index for the function Con, that is, let

$$\varphi_k(\langle m, x, y\rangle) = \mathrm{Con}(m, x, y).$$

Define s by

$$s(i, m, x) = c(i, c(k, c(R(m), R(x)))).$$

Then we have

$$\varphi_{s(i,m,x)}(y) = \varphi_i \circ \varphi_k \circ \varphi_{R(m)} \circ \varphi_{R(x)}(y) = \varphi_i(\mathrm{Con}(m, x, y)),$$

as desired. Notice that if the composition function c is primitive recursive, then s is also primitive recursive. In particular, for the specific indexing of the RAM programs given in Section 4.3, the function s is primitive recursive. $\square$

In practice, when using the s-m-n Theorem we usually denote the function $s(i, m, x)$ simply as $s(x)$.

As a first application of the s-m-n Theorem, we show that any two acceptable indexings are effectively inter-translatable, that is, computably inter-translatable.

Theorem 5.2. *Let $\varphi_0, \varphi_1, \ldots,$ be a universal indexing, and let $\psi_0, \psi_1, \ldots,$ be any indexing with a total computable s-1-1 function, that is, a function s such that*

$$\psi_{s(i,1,x)}(y) = \psi_i(x, y)$$

for all $i, x, y \in \mathbb{N}$. Then there is a total computable function t such that $\varphi_i = \psi_{t(i)}$.

Proof. Let φ_{univ} be a universal partial computable function for the indexing $\varphi_0, \varphi_1, \ldots$. Since $\psi_0, \psi_1, \ldots,$ is also an indexing φ_{univ} occurs somewhere in the second list, and thus, there is some k such that $\varphi_{univ} = \psi_k$. Then we have

$$\psi_{s(k,1,i)}(x) = \psi_k(i, x) = \varphi_{univ}(i, x) = \varphi_i(x),$$

for all $i, x \in \mathbb{N}$. Therefore, we can take the function t to be the function defined such that

$$t(i) = s(k, 1, i)$$

for all $i \in \mathbb{N}$. $\square$

Using Theorem 5.2, if we have two acceptable indexings $\varphi_0, \varphi_1, \ldots,$ and $\psi_0, \psi_1, \ldots$, there exist total computable functions t and u such that

$$\varphi_i = \psi_{t(i)} \quad \text{and} \quad \psi_i = \varphi_{u(i)}$$

for all $i \in \mathbb{N}$.

Also note that if the composition function c is primitive recursive, then any s-m-n function is primitive recursive, and the translation functions are primitive recursive. Actually, a stronger result can be shown. It can be shown that for any two acceptable indexings, there exist total computable *injective* and *surjective* translation functions. In other words, any two acceptable indexings are recursively isomorphic (Roger's isomorphism theorem); see Machtey and Young [36]. Next we turn to algorithmically unsolvable, or *undecidable*, problems.

5.2 Undecidable Problems

We saw in Section 4.3 that the halting problem for RAM programs is undecidable. In this section, we take a slightly more general approach to study the undecidability of problems, and give some tools for resolving decidability questions.

First, we prove again the undecidability of the halting problem, but this time, for *any* indexing of the partial computable functions.

Theorem 5.3. *(Halting Problem, Abstract Version) Let $\psi_0, \psi_1, \ldots,$ be any indexing of the partial computable functions. Then the function f defined such that*

$$f(x, y) = \begin{cases} 1 & \text{if } \psi_x(y) \text{ is defined,} \\ 0 & \text{if } \psi_x(y) \text{ is undefined,} \end{cases}$$

is not computable.

Proof. Assume that f is computable, and let g be the function defined such that

$$g(x) = f(x, x)$$

for all $x \in \mathbb{N}$. Then g is also computable. Let θ be the function defined such that

$$\theta(x) = \begin{cases} 0 & \text{if } g(x) = 0, \\ \text{undefined} & \text{if } g(x) = 1. \end{cases}$$

We claim that θ is not even partial computable. Observe that θ is such that

$$\theta(x) = \begin{cases} 0 & \text{if } \psi_x(x) \text{ is undefined,} \\ \text{undefined} & \text{if } \psi_x(x) \text{ is defined.} \end{cases}$$

If θ was partial computable, it would occur in the list as some ψ_i, and we would have

$$\theta(i) = \psi_i(i) = 0 \quad \text{iff} \quad \psi_i(i) \text{ is undefined,}$$

a contradiction. Therefore, f and g can't be computable. $\square$

Observe that the proof of Theorem 5.3 *does not* use the fact that the indexing is universal or acceptable, and thus, the theorem holds for *any indexing* of the partial computable functions.

Given any set, X, for any subset, $A \subseteq X$, of X, recall that the *characteristic function, C_A (or χ_A), of A* is the function, $C_A \colon X \to \{0, 1\}$, defined so that, for all $x \in X$,

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

The function g defined in the proof of Theorem 5.3 is the characteristic function of an important set denoted as K.

Definition 5.2. Given any indexing (ψ_i) of the partial computable functions, the set K is defined by

$$K = \{x \mid \psi_x(x) \text{ is defined}\}.$$

The set K is an abstract version of the halting problem. It is example of a set which is *not* computable (or not recursive). Since this fact is quite important, we give the following definition:

Definition 5.3. A subset A of Σ^* (or a subset A of $\mathbb{N}$) is *computable*, or *recursive*,[1] or *decidable* iff its characteristic function, C_A, is a total computable function.

Using Definition 5.3, Theorem 5.3 can be restated as follows.

Proposition 5.4. *For any indexing $\varphi_0, \varphi_1, \ldots$ of the partial computable functions (over Σ^* or $\mathbb{N}$), the set $K = \{x \mid \varphi_x(x) \text{ is defined}\}$ is not computable (not recursive).*

Computable (recursive) sets allow us to define the concept of a decidable (or undecidable) problem. The idea is to generalize the situation described in Section 4.3 and Section 4.6, where a set of objects, the RAM programs, is encoded into a set of natural numbers, using a coding scheme. For example, we would like to discuss the notion of computabilty of sets of trees or sets of graphs.

Definition 5.4. Let C be a countable set of objects, and let P be a property of objects in C. We view P as the set

$$\{a \in C \mid P(a)\}.$$

A *coding-scheme* is an injective function $\#\colon C \to \mathbb{N}$ that assigns a unique code to each object in C. The property P is *decidable (relative to $\#$)* iff the set $\{\#(a) \mid a \in C \text{ and } P(a)\}$ is computable (recursive). The property P is *undecidable (relative to $\#$)* iff the set $\{\#(a) \mid a \in C \text{ and } P(a)\}$ is not computable (not recursive).

Observe that the decidability of a property P of objects in C depends upon the coding scheme $\#$. Thus, if we are cheating in using a non-effective (*i.e* not computable by a computer program) coding scheme, we may declare that a property is decidable even though it is not decidable in some reasonable coding scheme. Consequently, we require a coding scheme $\#$ to be *effective* in the following sense. Given any object $a \in C$, we can effectively (*i.e.* algorithmically) determine its code $\#(a)$. Conversely, given any integer $n \in \mathbb{N}$, we should be able to tell effectively if n is the code of some object in C, and if so, to find this object. In practice, it is always possible to describe the objects in C as strings over some (possibly complex) alphabet Σ (sets of trees, graphs, etc). In such cases, the coding schemes are computable functions from Σ^* to $\mathbb{N} = \{a_1\}^*$.

For example, let $C = \mathbb{N} \times \mathbb{N}$, where the property P is the equality of the partial functions φ_x and φ_y. We can use the pairing function $\langle -, - \rangle$ as a coding function, and the problem is formally encoded as the computability (recursiveness) of the set

$$\{\langle x, y \rangle \mid x, y \in \mathbb{N},\, \varphi_x = \varphi_y\}.$$

In most cases, we don't even bother to describe the coding scheme explicitly, knowing that such a description is routine, although perhaps tedious.

We now show that most properties about programs (except the trivial ones) are undecidable.

[1]Since 1996, the term *recursive* has been considered old-fashioned by many researchers, and the term *computable* has been used instead.

5.3 Reducibility and Rice's Theorem

First, we show that it is undecidable whether a RAM program halts for every input. In other words, it is undecidable whether a procedure is an algorithm. We actually prove a more general fact.

Proposition 5.5. *For any acceptable indexing $\varphi_0, \varphi_1, \ldots$ of the partial computable functions, the set*

$$\mathrm{TOTAL} = \{x \mid \varphi_x \textit{ is a total function}\}$$

is not computable (not recursive).

Proof. The proof uses a technique known as reducibility. We try to reduce a set A known to be *noncomputable* (*nonrecursive*) to TOTAL via a computable function $f\colon A \to \mathrm{TOTAL}$, so that

$$x \in A \quad \text{iff} \quad f(x) \in \mathrm{TOTAL}.$$

If TOTAL were computable (recursive), its characteristic function g would be computable, and thus, the function $g \circ f$ would be computable, a contradiction, since A is assumed to be noncomputable (nonrecursive). In the present case, we pick $A = K$. To find the computable function $f\colon K \to \mathrm{TOTAL}$, we use the s-m-n Theorem. Let θ be the function defined below: for all $x, y \in \mathbb{N}$,

$$\theta(x,y) = \begin{cases} \varphi_x(x) & \text{if } x \in K, \\ \text{undefined} & \text{if } x \notin K. \end{cases}$$

Note that θ does not depend on y. The function θ is partial computable. Indeed, we have

$$\theta(x,y) = \varphi_x(x) = \varphi_{univ}(x,x).$$

Thus, θ has some index j, so that $\theta = \varphi_j$, and by the s-m-n Theorem, we have

$$\varphi_{s(j,1,x)}(y) = \varphi_j(x,y) = \theta(x,y).$$

Let f be the computable function defined such that

$$f(x) = s(j,1,x)$$

for all $x \in \mathbb{N}$. Then we have

$$\varphi_{f(x)}(y) = \begin{cases} \varphi_x(x) & \text{if } x \in K, \\ \text{undefined} & \text{if } x \notin K \end{cases}$$

for all $y \in \mathbb{N}$. Thus, observe that $\varphi_{f(x)}$ is a total function iff $x \in K$, that is,

$$x \in K \quad \text{iff} \quad f(x) \in \mathrm{TOTAL},$$

where f is computable. As we explained earlier, this shows that TOTAL is not computable (not recursive). $\square$

The above argument can be generalized to yield a result known as Rice's theorem. Let $\varphi_0, \varphi_1, \ldots$ be any indexing of the partial computable functions, and let C be any set of partial computable functions. We define the set P_C as

$$P_C = \{x \in \mathbb{N} \mid \varphi_x \in C\}.$$

We can view C as a property of some of the partial computable functions. For example

$$C = \{\text{all total computable functions}\}.$$

We say that C is *nontrivial* if C is neither empty nor the set of all partial computable functions. Equivalently C is nontrivial iff $P_C \neq \emptyset$ and $P_C \neq \mathbb{N}$. We may think of P_C as the set of programs computing the functions in C.

Theorem 5.6. *(Rice's Theorem, 1953) For any acceptable indexing $\varphi_0, \varphi_1, \ldots$ of the partial computable functions, for any set C of partial computable functions, the set*

$$P_C = \{x \in \mathbb{N} \mid \varphi_x \in C\}$$

is not computable (not recursive) unless C is trivial.

Proof. Assume that C is nontrivial. A set is computable (recursive) iff its complement is computable (recursive) (the proof is trivial). Hence, we may assume that the totally undefined function is not in C, and since $C \neq \emptyset$, let ψ be some other function in C. We produce a computable function f such that

$$\varphi_{f(x)}(y) = \begin{cases} \psi(y) & \text{if } x \in K, \\ \text{undefined} & \text{if } x \notin K, \end{cases}$$

for all $y \in \mathbb{N}$. We get f by using the s-m-n Theorem. Let $\psi = \varphi_i$, and define θ as follows:

$$\theta(x, y) = \varphi_{univ}(i, y) + (\varphi_{univ}(x, x) \dot{-} \varphi_{univ}(x, x)),$$

where $\dot{-}$ is the primitive recursive function monus for truncated subtraction; see Section 3.7. Recall that $\varphi_{univ}(x, x) \dot{-} \varphi_{univ}(x, x)$ is defined iff $\varphi_{univ}(x, x)$ is defined iff $x \in K$, and so

$$\theta(x, y) = \varphi_{univ}(i, y) = \varphi_i(y) = \psi(y) \quad \text{iff} \quad x \in K$$

and $\theta(x, y)$ is undefined otherwise. Clearly θ is partial computable, and we let $\theta = \varphi_j$. By the s-m-n Theorem, we have

$$\varphi_{s(j,1,x)}(y) = \varphi_j(x, y) = \theta(x, y)$$

for all $x, y \in \mathbb{N}$. Letting f be the computable function such that

$$f(x) = s(j, 1, x),$$

by definition of θ, we get

$$\varphi_{f(x)}(y) = \theta(x, y) = \begin{cases} \psi(y) & \text{if } x \in K, \\ \text{undefined} & \text{if } x \notin K. \end{cases}$$

Thus, f is the desired reduction function. Now we have

$$x \in K \quad \text{iff} \quad f(x) \in P_C,$$

and thus, the characteristic function C_K of K is equal to $C_P \circ f$, where C_P is the characteristic function of P_C. Therefore, P_C is not computable (not recursive), since otherwise, K would be computable, a contradiction. $\square$

Rice's theorem shows that all nontrivial properties of the input/output behavior of programs are undecidable!

It is important to understand that Rice's theorem says that *the set P_C of indices of all partial computable functions equal to some function in a given set C of partial computable functions is not computable* if C is nontrivial, *not that the set C is not computable* if C is nontrivial. The second statement does not make any sense because our machinery only applies to sets of natural numbers (or sets of strings). For example, the set $C = \{\varphi_{i_0}\}$ consisting of a single partial computable function is nontrivial, and being finite, under the second wrong interpretation it would be computable. But we need to consider the set

$$P_C = \{n \in \mathbb{N} \mid \varphi_n = \varphi_{i_0}\}$$

of indices of all partial computable functions φ_n that are equal to φ_{i_0}, and by Rice's theorem, this set is not computable. In other words, it is undecidable whether an arbitrary partial computable function is equal to some fixed partial computable function.

The scenario to apply Rice's theorem to a class C of partial functions is to show that *some partial computable function belongs to C* (C is not empty), *and that some partial computable function does not belong to C* (C is not all the partial computable functions). This demonstrates that C is nontrivial.

In particular, the following properties are undecidable.

Proposition 5.7. *The following properties of partial computable functions are undecidable.*

(a) A partial computable function is a constant function.

(b) Given any integer $y \in \mathbb{N}$, is y in the range of some partial computable function.

(c) Two partial computable functions φ_x and φ_y are identical. More precisely, the set $\{\langle x, y\rangle \mid \varphi_x = \varphi_y\}$ is not computable.

(d) A partial computable function φ_x is equal to a given partial computable function φ_a.

(e) A partial computable function yields output z on input y, for any given $y, z \in \mathbb{N}$.

(f) A partial computable function diverges for some input.

(g) A partial computable function diverges for all input.

The above proposition is left as an easy exercise. For example, in (a), we need to exhibit a constant (partial) computable function, such as $zero(n) = 0$, and a nonconstant (partial) computable function, such as the identity function (or $succ(n) = n + 1$).

A property may be undecidable although it is partially decidable. By partially decidable, we mean that there exists a computable function g that enumerates the set $P_C = \{x \mid \varphi_x \in C\}$. This means that there is a computable function g whose range is P_C. We say that P_C is *listable*, or *computably enumerable*, or *recursively enumerable*. Indeed, g provides a recursive enumeration of P_C, with possible repetitions. Listable sets are the object of the next section.

5.4 Listable (Recursively Enumerable) Sets

In this section and the next our focus is on subsets of $\mathbb{N}$ rather than on numerical functions. Consider the set

$$A = \{k \in \mathbb{N} \mid \varphi_k(a) \text{ is defined}\},$$

where $a \in \mathbb{N}$ is any fixed natural number. By Rice's theorem, A is not computable (not recursive); check this. We claim that A is the range of a computable function g. For this, we use the T-predicate introduced in Definition 4.13. Recall that the predicate $T(i, y, z)$ is defined as follows:

$T(i, y, z)$ holds iff i codes a RAM program, y is an input, and z codes a halting computation of program P_i on input y.

We produce a function which is actually primitive recursive. First, note that A is nonempty (why?), and let x_0 be any index in A. We define g by primitive recursion as follows:

$$\begin{aligned} g(0) &= x_0, \\ g(x+1) &= \begin{cases} \Pi_1(x) & \text{if } T(\Pi_1(x), a, \Pi_2(x)), \\ x_0 & \text{otherwise.} \end{cases} \end{aligned}$$

Since this type of argument is new, it is helpful to explain informally what g does. For every input x, the function g tries finitely many steps of a computation on input a for some partial computable function φ_i computed by the RAM program P_i. Since we need to consider all pairs (i, z) but we only have one variable x at our disposal, we use the trick of packing i and z into $x = \langle i, z \rangle$. Then the index i of the partial function is given by $i = \Pi_1(x)$ and the guess for the code of the computation is given by $z = \Pi_2(x)$. Since Π_1 and Π_2 are projection functions, when x ranges over $\mathbb{N}$, both $i = \Pi_1(x)$ and $z = \Pi_2(x)$ also range over $\mathbb{N}$. Thus *every* partial function φ_i and *every* code for a computation z will be tried, and whenever $\varphi_i(a)$ is defined, which means that there is a correct guess for the code z of the halting computation of P_i on input a, $T(\Pi_1(x), a, \Pi_2(x)) = T(i, a, z)$ is true, and $g(x+1)$ returns i.

Such a process is called a *dovetailing* computation. This type of argument will be used over and over again.

Definition 5.5. A subset X of $\mathbb{N}$ is *listable*, or *computably enumerable*, or *recursively enumerable*[2] iff either $X = \emptyset$, or X is the range of some total computable function (total recursive function). Similarly, a subset X of Σ^* is *listable* or *computably enumerable*, or *recursively enumerable* iff either $X = \emptyset$, or X is the range of some total computable function (total recursive function).

We will often abbreviate computably enumerable as *c.e*, (and recursively enumerable as *r.e.*). A computably enumerable set is sometimes called a *partially decidable* or *semidecidable* set.

Remark: It should be noted that the definition of a *listable set* (*c.e set* or *r.e. set*) given in Definition 5.5 is *different* from an earlier definition given in terms of acceptance by a Turing machine and it is by no means obvious that these two definitions are equivalent. This equivalence will be proven in Proposition 5.9 ((1) $\Longleftrightarrow$ (4)).

The following proposition relates computable sets and listable sets (recursive sets and recursively enumerable sets).

Proposition 5.8. *A set A is computable (recursive) iff both A and its complement $\overline{A}$ are listable (computably enumerable, recursively enumerable).*

Proof. Assume that A is computable. Then it is trivial that its complement is also computable. Hence, we only have to show that a computable set is listable. The empty set is listable by definition. Otherwise, let $y \in A$ be any element. Then the function f defined such that

$$f(x) = \begin{cases} x & \text{iff } C_A(x) = 1, \\ y & \text{iff } C_A(x) = 0, \end{cases}$$

for all $x \in \mathbb{N}$ is computable and has range A.

Conversely, assume that both A and $\overline{A}$ are listable. If either A or $\overline{A}$ is empty, then A is computable. Otherwise, let $A = f(\mathbb{N})$ and $\overline{A} = g(\mathbb{N})$, for some computable functions f and g. We define the function C_A as follows:

$$C_A(x) = \begin{cases} 1 & \text{if } f(\min y[f(y) = x \ \vee \ g(y) = x]) = x, \\ 0 & \text{otherwise.} \end{cases}$$

The function C_A lists A and $\overline{A}$ in parallel, waiting to see whether x turns up in A or in $\overline{A}$. Note that x must eventually turn up either in A or in $\overline{A}$, so that C_A is a total computable function. $\square$

[2]Since 1996, the term *recursively enumerable* has been considered old-fashioned by many researchers, and the terms *listable* and *computably enumerable* have been used instead.

Our next goal is to show that the listable (recursively enumerable) sets can be given several equivalent definitions.

Proposition 5.9. *For any subset A of $\mathbb{N}$, the following properties are equivalent:*

(1) A is empty or A is the range of a primitive recursive function (Rosser, 1936).

(2) A is listable (computably enumerable, recursively enumerable).

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

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

Proof. The implication (1) $\Rightarrow$ (2) is trivial, since A is listable iff either it is empty or it is the range of a (total) computable function.

To prove the implication (2) $\Rightarrow$ (3), it suffices to observe that the empty set is the range of the totally undefined function (computed by an infinite loop program), and that a computable function is a partial computable function.

The implication (3) $\Rightarrow$ (4) is shown as follows. Assume that A is the range of φ_i. Define the function f such that

$$f(x) = \min k[T(i, \Pi_1(k), \Pi_2(k)) \ \wedge \ \mathrm{Res}(\Pi_2(k)) = x]$$

for all $x \in \mathbb{N}$. Since $A = \varphi_i(\mathbb{N})$, we have $x \in A$ iff there is some input $y \in \mathbb{N}$ and some computation coded by z such that the RAM program P_i on input y has a halting computation coded by z and produces the output x. Using the T-predicate, this is equivalent to $T(i, y, z)$ and $\mathrm{Res}(z) = x$. Since we need to search over all pairs (y, z), we pack y and z as $k = \langle y, z\rangle$ so that $y = \Pi_1(k)$ and $z = \Pi_2(k)$, and we search over all $k \in \mathbb{N}$. If the search succeeds, which means that $T(i, y, z)$ and $\mathrm{Res}(z) = x$, we set $f(x) = k = \langle y, z\rangle$, so that f is a function whose domain in the range of φ_i (namely A). Note that the value $f(x)$ is irrelevant, but it is convenient to pick k. Clearly, f is partial computable and has domain A.

The implication (4) $\Rightarrow$ (1) is shown as follows. The only nontrivial case is when A is nonempty. Assume that A is the domain of φ_i. Since $A \neq \emptyset$, there is some $a \in \mathbb{N}$ such that $a \in A$, which means that for some input y the RAM program P_i has a halting computation coded by z on input a, so if we pack y and z as $k = \langle y, z\rangle$, the quantity

$$\min k[T(i, \Pi_1(k), \Pi_2(k))] = \min\langle y, z\rangle[T(i, y, z)]$$

is defined. We can pick a to be

$$a = \Pi_1(\min k[T(i, \Pi_1(k), \Pi_2(k))]).$$

We define the primitive recursive function f as follows:

$$\begin{aligned} f(0) &= a, \\ f(x+1) &= \begin{cases} \Pi_1(x) & \text{if } T(i, \Pi_1(x), \Pi_2(x)), \\ a & \text{if } \neg T(i, \Pi_1(x), \Pi_2(x)). \end{cases} \end{aligned}$$

Some $y \in \mathbb{N}$ is in the domain of φ_i (namely A) iff the RAM program P_i has a halting computation coded by z on input y iff $T(i, y, z)$ is true. If we pack y and z as $x = \langle y, z\rangle$, then $T(i, y, z) = T(i, \Pi_1(x), \Pi_2(x))$, so if we search over all $x = \langle y, z\rangle$ we search over all y and all z. Whenever $T(i, y, z) = T(i, \Pi_1(x), \Pi_2(x))$ holds, we set $f(x + 1) = y$ since $y \in A$, and if $T(i, y, z) = T(i, \Pi_1(x), \Pi_2(x))$ is false, we return the default value $a \in A$. Our search will find all y such that $T(i, y, z) = T(i, \Pi_1(x), \Pi_2(x))$ holds for some z, which means that all $y \in A$ will be in the range of f. By construction, f only has values in A. Clearly, f is primitive recursive. $\square$

More intuitive proofs of the implications (3) $\Rightarrow$ (4) and (4) $\Rightarrow$ (1) can be given. Assume that $A \neq \emptyset$ and that $A = range(g)$, where g is a partial computable function. Assume that g is computed by a RAM program P. To compute $f(x)$, we start computing the sequence

$$g(0), g(1), \ldots$$

looking for x. If x turns up as say $g(n)$, then we output n. Otherwise the computation diverges. Hence, the domain of f is the range of g.

Assume now that A is the domain of some partial computable function g, and that g is computed by some Turing machine M. Since the case where $A = \emptyset$ is trivial, we may assume that $A \neq \emptyset$, and let $n_0 \in A$ be some chosen element in A. We construct another Turing machine performing the following steps: On input n,

(0) Do one step of the computation of $g(0)$

$\ldots$

(n) Do $n + 1$ steps of the computation of $g(0)$

Do n steps of the computation of $g(1)$

$\ldots$

Do 2 steps of the computation of $g(n - 1)$

Do 1 step of the computation of $g(n)$

During this process, whenever the computation of $g(m)$ halts for some $m \leq n$, we output m. Otherwise, we output n_0.

In this fashion, we will enumerate the domain of g, and since we have constructed a Turing machine that halts for every input, we have a total computable function.

The following proposition can easily be shown using the proof technique of Proposition 5.9.

Proposition 5.10. *The following facts hold.*

(1) There is a computable function h such that

$$range(\varphi_x) = dom(\varphi_{h(x)}) \quad \textit{for all } x \in \mathbb{N}.$$

(2) There is a computable function k such that

$$dom(\varphi_x) = range(\varphi_{k(x)})$$

and $\varphi_{k(x)}$ is total computable, for all $x \in \mathbb{N}$ such that $dom(\varphi_x) \neq \emptyset$.

The proof of Proposition 5.10 is left as an exercise.

Using Proposition 5.9, we can prove that K is a listable set. Indeed, we have $K = dom(f)$, where

$$f(x) = \varphi_{univ}(x, x) \quad \text{for all } x \in \mathbb{N}.$$

The set

$$K_0 = \{\langle x, y\rangle \mid \varphi_x(y) \text{ is defined}\}$$

is also a listable set, since $K_0 = dom(g)$, where

$$g(z) = \varphi_{univ}(\Pi_1(z), \Pi_2(z)),$$

which is partial computable. It worth recording these facts in the following lemma.

Proposition 5.11. *The sets K and K_0 are listable (c.e., r.e.) sets that are not computable sets (not recursive).*

We can now prove that there are sets that are not listable (not c.e., not r.e.).

Proposition 5.12. *For any indexing of the partial computable functions, the complement $\overline{K}$ of the set*

$$K = \{x \in \mathbb{N} \mid \varphi_x(x) \text{ is defined}\}$$

is not listable (not computably enumerable, not recursively enumerable).

Proof. If $\overline{K}$ was listable, since K is also listable, by Proposition 5.8, the set K would be computable, a contradiction. $\square$

The sets $\overline{K}$ and $\overline{K_0}$ are examples of sets that are not listable (not c.e., not r.e.). This shows that the listable (c.e., r.e.) sets are not closed under complementation. However, we leave it as an exercise to prove that the listable (c.e., r.e.) sets are closed under union and intersection.

We will prove later on that TOTAL is not listable (not c.e., not r.e.). This is rather unpleasant. Indeed, this means that there is no way of effectively listing all algorithms (all total computable functions). Hence, in a certain sense, the concept of partial computable function (procedure) is more natural than the concept of a (total) computable function (algorithm).

The next two propositions give other characterizations of the listable (c.e., r.e. sets) and of the computable sets (recursive sets). The proofs are left as an exercise.

Proposition 5.13. *The following facts hold.*

(1) A set A is listable (c.e., r.e.) iff either it is finite or it is the range of an injective computable function.

(2) A set A is listable (c.e., r.e.) if either it is empty or it is the range of a monotonic partial computable function.

(3) A set A is listable (c.e., r.e.) iff there is a Turing machine M such that, for all $x \in \mathbb{N}$, M halts on x iff $x \in A$.

Proposition 5.14. *A set A is computable (recursive) iff either it is finite or it is the range of a strictly increasing computable function.*

Another important result relating the concept of partial computable function and that of a listable (c.e., r.e.) set is given below.

Theorem 5.15. *For every unary partial function f, the following properties are equivalent:*

(1) f is partial computable.

(2) The set

$$\{\langle x, f(x)\rangle \mid x \in dom(f)\}$$

is listable (c.e., r.e.).

Proof. Let $g(x) = \langle x, f(x)\rangle$. Clearly, g is partial computable, and

$$range(g) = \{\langle x, f(x)\rangle \mid x \in dom(f)\}.$$

Conversely, assume that

$$range(g) = \{\langle x, f(x)\rangle \mid x \in dom(f)\}$$

for some computable function g. Then we have

$$f(x) = \Pi_2(g(\min y[\Pi_1(g(y)) = x])) \quad \text{for all } x \in \mathbb{N},$$

so that f is partial computable. ☐

Using our indexing of the partial computable functions and Proposition 5.9, we obtain an indexing of the listable (c.e., r.e.) sets.

Definition 5.6. For any acceptable indexing $\varphi_0, \varphi_1, \ldots$ of the partial computable functions, we define the enumeration $W_0, W_1, \ldots$ of the listable (c.e., r.e.) sets by setting

$$W_x = dom(\varphi_x).$$

We now describe a technique for showing that certain sets are listable (c.e., r.e.) but not computable (not recursive), or complements of listable (c.e., r.e.) sets that are not computable (not recursive), or not listable (not c.e., not r.e.), or neither listable (not c.e., not r.e.) nor the complement of a listable (c.e., r.e.) set. This technique is known as *reducibility*.

5.5 Reducibility and Complete Sets

We already used the notion of reducibility in the proof of Proposition 5.5 to show that TOTAL is not computable (not recursive).

Definition 5.7. Let A and B be subsets of $\mathbb{N}$ (or Σ^*). We say that the set A is *many-one reducible* to the set B if there is a *total computable* function (or *total recursive* function) $f\colon \mathbb{N} \to \mathbb{N}$ (or $f\colon \Sigma^* \to \Sigma^*$) such that

$$x \in A \quad \text{iff} \quad f(x) \in B \quad \text{for all } x \in \mathbb{N}.$$

We write $A \leq B$, and for short, we say that A is *reducible* to B. Sometimes, the notation $A \leq_m B$ is used to stress that this is a many-to-one reduction (that is, f is not necessarily injective).

Intuitively, deciding membership in B is as hard as deciding membership in A. This is because any method for deciding membership in B can be converted to a method for deciding membership in A by first applying f to the number (or string) to be tested.

Remark: Besides many-to-one reducibility, there is a also a notion of *one-one reducibility* defined as follows: the set A is *one-one reducible* to the set B if there is a *total* **injective** *computable* function $f\colon \mathbb{N} \to \mathbb{N}$ such that

$$x \in A \quad \text{iff} \quad f(x) \in B \quad \text{for all } x \in \mathbb{N}.$$

We write $A \leq_1 B$. Obviously $A \leq_1 B$ implies $A \leq_m B$ so one-one reducibiity is a stronger notion. We do not need one-one reducibility for our purposes so we will not discuss it. We refer the interested reader to Rogers [44] (especially Chapter 7) for more on reducibility.

The following simple proposition is left as an exercise to the reader.

Proposition 5.16. *Let A, B, C be subsets of $\mathbb{N}$ (or Σ^*). The following properties hold:*

(1) If $A \leq B$ and $B \leq C$, then $A \leq C$.

(2) If $A \leq B$ then $\overline{A} \leq \overline{B}$.

(3) If $A \leq B$ and B is listable (c.e., r.e.), then A is listable (c.e., r.e.).

(4) If $A \leq B$ and A is not listable (not c.e., not r.e.), then B is not listable (not c.e., not r.e.).

(5) If $A \leq B$ and B is computable, then A is computable.

(6) If $A \leq B$ and A is not computable, then B is not computable.

Part (4) of Proposition 5.16 is often useful for proving that some set B is not listable. It suffices to reduce some set known to be nonlistable to B, for example $\overline{K}$. Similarly, Part (6) of Proposition 5.16 is often useful for proving that some set B is not computable. It suffices to reduce some set known to be noncomputable to B, for example K.

Observe that $A \leq B$ implies that $\overline{A} \leq \overline{B}$, but *not* that $\overline{B} \leq \overline{A}$.

Part (3) of Proposition 5.16 may be useful for proving that some set A is listable. It suffices to reduce A to some set known to be listable, for example K. Similarly, Part (5) of Proposition 5.16 may be useful for proving that some set A is computable. It suffices to reduce A to some set known to be computable. In practice, it is often easier to prove directly that A is computable by showing that both A and $\overline{A}$ are listable.

Another important concept is the concept of a complete set.

Definition 5.8. A listable (c.e., r.e.) set A is *complete w.r.t. many-one reducibility* iff every listable (c.e., r.e.) set B is reducible to A, *i.e.*, $B \leq A$.

For simplicity, we will often say *complete* for *complete w.r.t. many-one reducibility*. Intuitively, a complete listable (c.e., r.e.) set is a "hardest" listable (c.e., r.e.) set as far as membership is concerned.

Theorem 5.17. *The following properties hold:*

(1) If A is complete, B is listable (c.e, r.e.), and $A \leq B$, then B is complete.

(2) K_0 is complete.

(3) K_0 is reducible to K. Consequently, K is also complete.

Proof. (1) This is left as a simple exercise.

(2) Let W_x be any listable set (recall Definition 5.6). Then

$$y \in W_x \quad \text{iff} \quad \langle x, y \rangle \in K_0,$$

and the reduction function is the computable function f such that

$$f(y) = \langle x, y \rangle \quad \text{for all } y \in \mathbb{N}.$$

(3) We use the s-m-n Theorem. First, we leave it as an exercise to prove that there is a computable function f such that

$$\varphi_{f(x)}(y) = \begin{cases} 1 & \text{if } \varphi_{\Pi_1(x)}(\Pi_2(x)) \text{ is defined,} \\ \text{undefined} & \text{otherwise,} \end{cases}$$

for all $x, y \in \mathbb{N}$. Then for every $z \in \mathbb{N}$,

$$z \in K_0 \quad \text{iff} \quad \varphi_{\Pi_1(z)}(\Pi_2(z)) \text{ is defined,}$$

iff $\varphi_{f(z)}(y) = 1$ for all $y \in \mathbb{N}$. However,

$$\varphi_{f(z)}(y) = 1 \quad \text{iff} \quad \varphi_{f(z)}(f(z)) = 1,$$

since $\varphi_{f(z)}$ is a constant function. This means that

$$z \in K_0 \quad \text{iff} \quad f(z) \in K,$$

and f is the desired function. $\square$

As a corollary of Theorem 5.17, the set K is also complete.

Definition 5.9. Two sets A and B have the same *degree of unsolvability* or are *equivalent* iff $A \leq B$ and $B \leq A$.

Since K and K_0 are both complete, they have the same degree of unsolvability in the set of listable sets.

We will now investigate the reducibility and equivalence of various sets.

Recall that

$$\text{TOTAL} = \{x \in \mathbb{N} \mid \varphi_x \text{ is total}\}.$$

We define EMPTY and FINITE, as follows:

$$\begin{aligned} \text{EMPTY} &= \{x \in \mathbb{N} \mid \varphi_x \text{ is undefined for all input}\}, \\ \text{FINITE} &= \{x \in \mathbb{N} \mid \varphi_x \text{ is defined only for finitely many input}\}. \end{aligned}$$

Obviously, EMPTY $\subset$ FINITE, and since

$$\text{FINITE} = \{x \in \mathbb{N} \mid \varphi_x \text{ has a finite domain}\},$$

we have

$$\overline{\text{FINITE}} = \{x \in \mathbb{N} \mid \varphi_x \text{ has an infinite domain}\},$$

and thus, TOTAL $\subset \overline{\text{FINITE}}$. Since

$$\text{EMPTY} = \{x \in \mathbb{N} \mid \varphi_x \text{ is undefined for all input}\}$$

we have

$$\overline{\text{EMPTY}} = \{x \in \mathbb{N} \mid \varphi_x \text{ is defined for some input}\},$$

we have $\overline{\text{FINITE}} \subseteq \overline{\text{EMPTY}}$.

Proposition 5.18. *We have* $K_0 \leq \overline{\text{EMPTY}}$.

The proof of Proposition 5.18 follows from the proof of Theorem 5.17. We also have the following proposition.

Proposition 5.19. *The following properties hold:*

(1) EMPTY *is not listable (not c.e., not r.e.).*

(2) $\overline{\text{EMPTY}}$ *is listable (c.e., r.e.).*

(3) $\overline{K}$ *and* EMPTY *are equivalent.*

(4) $\overline{\text{EMPTY}}$ *is complete.*

Proof. We prove (1) and (3), leaving (2) and (4) as an exercise (Actually, (2) and (4) follow easily from (3)). First, we show that $\overline{K} \leq$ EMPTY. By the s-m-n Theorem, there exists a computable function f such that

$$\varphi_{f(x)}(y) = \begin{cases} \varphi_x(x) & \text{if } \varphi_x(x) \text{ is defined,} \\ \text{undefined} & \text{if } \varphi_x(x) \text{ is undefined,} \end{cases}$$

for all $x, y \in \mathbb{N}$. Note that for all $x \in \mathbb{N}$,

$$x \in \overline{K} \quad \text{iff} \quad f(x) \in \text{EMPTY},$$

and thus, $\overline{K} \leq$ EMPTY. Since $\overline{K}$ is not listable, EMPTY is not listable.

We now prove (3). By the s-m-n Theorem, there is a computable function g such that

$$\varphi_{g(x)}(y) = \min z[T(x, \Pi_1(z), \Pi_2(z))], \quad \text{for all } x, y \in \mathbb{N}.$$

Note that

$$x \in \text{EMPTY} \quad \text{iff} \quad g(x) \in \overline{K} \quad \text{for all } x \in \mathbb{N}.$$

Therefore, EMPTY $\leq \overline{K}$, and since we just showed that $\overline{K} \leq$ EMPTY, the sets $\overline{K}$ and EMPTY are equivalent. ☐

Proposition 5.20. *The following properties hold:*

(1) TOTAL *and* $\overline{\text{TOTAL}}$ *are not listable (not c.e., not r.e.).*

(2) FINITE *and* $\overline{\text{FINITE}}$ *are not listable (not c.e, not r.e.).*

Proof. Checking the proof of Theorem 5.17, we note that $K_0 \leq$ TOTAL and $K_0 \leq \overline{\text{FINITE}}$. Hence, we get $\overline{K_0} \leq \overline{\text{TOTAL}}$ and $\overline{K_0} \leq$ FINITE, and neither $\overline{\text{TOTAL}}$ nor FINITE is listable. If TOTAL was listable, then there would be a computable function f such that TOTAL $= range(f)$. Define g as follows:

$$g(x) = \varphi_{f(x)}(x) + 1 = \varphi_{univ}(f(x), x) + 1$$

for all $x \in \mathbb{N}$. Since f is total and $\varphi_{f(x)}$ is total for all $x \in \mathbb{N}$, the function g is total computable. Let e be an index such that

$$g = \varphi_{f(e)}.$$

Since g is total, $g(e)$ is defined. Then we have

$$g(e) = \varphi_{f(e)}(e) + 1 = g(e) + 1,$$

a contradiction. Hence, TOTAL is not listable. Finally, we show that TOTAL $\leq \overline{\text{FINITE}}$. This also shows that $\overline{\text{FINITE}}$ is not listable. By the s-m-n Theorem, there is a computable function f such that

$$\varphi_{f(x)}(y) = \begin{cases} 1 & \text{if } \forall z \leq y(\varphi_x(z)\downarrow), \\ \text{undefined} & \text{otherwise,} \end{cases}$$

for all $x, y \in \mathbb{N}$. It is easily seen that

$$x \in \text{TOTAL} \quad \text{iff} \quad f(x) \in \overline{\text{FINITE}} \quad \text{for all } x \in \mathbb{N}. \qquad \square$$

From Proposition 5.20, we have TOTAL $\leq \overline{\text{FINITE}}$. It turns out that $\overline{\text{FINITE}} \leq$ TOTAL, and TOTAL and $\overline{\text{FINITE}}$ are equivalent.

Proposition 5.21. *The sets* TOTAL *and* $\overline{\text{FINITE}}$ *are equivalent.*

Proof. We show that $\overline{\text{FINITE}} \leq$ TOTAL. By the s-m-n Theorem, there is a computable function f such that

$$\varphi_{f(x)}(y) = \begin{cases} 1 & \text{if } \exists z \geq y(\varphi_x(z)\downarrow), \\ \text{undefined} & \text{if } \forall z \geq y(\varphi_x(z)\uparrow), \end{cases}$$

for all $x, y \in \mathbb{N}$. It is easily seen that

$$x \in \overline{\text{FINITE}} \quad \text{iff} \quad f(x) \in \text{TOTAL} \quad \text{for all } x \in \mathbb{N}. \qquad \square$$

More advanced topics such that the recursion theorem, the extended Rice Theorem, and creative and productive sets will be discussed in Chapter 7.

Chapter 6

The Lambda-Calculus

The original motivation of Alonzo Church for inventing the λ-calculus was to provide a type-free foundation for mathematics (alternate to set theory) based on higher-order logic and the notion of *function* in the early 1930's (1932,1933) This attempt to provide such a foundation for mathematics failed due to a form of Russell's paradox. Church was clever enough to turn the technical reason for this failure, the existence of fixed-point combinators, into a success, namely to view the λ-calculus as a formalism for defining the notion of *computability* (1932,1933,1935). The λ-calculus is indeed one of the first computation models, slightly preceding the Turing machine.

Kleene proved in 1936 that all the computable functions (recursive functions) in the sense of Herbrand and Gödel are definable in the λ-calculus, showing that the λ-calculus has *universal computing power*. In 1937, Turing proved that Turing machines compute the same class of computable functions. (This paper is very hard to read, in part because the definition of a Turing machine is not included in this paper). In short, the λ-calculus and Turing machines have *the same computing power*. Here we have to be careful. To be precise we should have said that all the *total* computable functions (total recursive functions) are definable in the λ-calculus. In fact, it is also true that all the *partial* computable functions (partial recursive functions) are definable in the λ-calculus but this requires more care.

Since the λ-calculus does not have any notion of tape, register, or any other means of storing data, it quite amazing that the λ-calculus has so much computing power.

The λ-calculus is based on three concepts:

(1) Application.

(2) Abstraction (also called λ-abstraction).

(3) β-reduction (and β-conversion).

If f is a function, say the exponential function $f\colon \mathbb{N} \to \mathbb{N}$ given by $f(n) = 2^n$, and if n a natural number, then the result of applying f to a natural number, say 5, is written as

$$(f5)$$

and is called an *application*. Here we can agree that f and 5 do not have the same *type*, in the sense that f is a function and 5 is a number, so applications such as (ff) or (55) do not make sense, but the λ-calculus is *type-free* so expressions such as (ff) as allowed. This may seem silly, and even possibly undesirable, but allowing self application turns out to a major reason for the computing power of the λ-calculus.

Given an expression M containing a variable x, say

$$M(x) = x^2 + x + 1,$$

as x ranges over $\mathbb{N}$, we obtain the function respresented in standard mathematical notation by $x \mapsto x^2 + x + 1$. If we supply the input value 5 for x, then the value of the function is $5^2 + 5 + 1 = 31$. Church introduced the notation

$$\lambda x.\, (x^2 + x + 1)$$

for this function. Here, we have an *abstraction*, in the sense that the static expression $M(x)$ for x fixed becomes an "abstract" function denoted $\lambda x.\, M$.

It would be pointless to only have the two concepts of application and abstraction. The glue between these two notions is a form of evaluation called *β-reduction*.[1] Given a λ-abstraction $\lambda x.\, M$ and some other term N (thought of as an argument), we have the "evaluation" rule, we say *β-reduction*,

$$(\lambda x.\, M)N \xrightarrow{+}_\beta M[x := N],$$

where $M[x := N]$ denotes the result of substituting N for all occurrences of x in M. For example, if $M = \lambda x.\, (x^2 + x + 1)$ and $N = 2y + 1$, we have

$$(\lambda x.\, (x^2 + x + 1))(2y + 1) \xrightarrow{+}_\beta (2y + 1)^2 + 2y + 1 + 1.$$

Observe that β-reduction is a *purely formal* operation (plugging N wherever x occurs in M), and that the expression $(2y+1)^2+2y+1+1$ is *not* instantly simplified to $4y^2+6y+3$. In the λ-calculus, the natural numbers as well as the arithmetic operations $+$ and $\times$ need to be represented as λ-terms in such a way that they "evaluate" correctly using only β-conversion. In this sense, the λ-calculus is an incredibly low-level programming language. Nevertheless, the λ-calculus is the core of various *functional programming languages* such as *OCaml*, *ML*, *Miranda* and *Haskell*, among others.

We now proceed with precise definitions and results. But first we ask the reader not to think of functions as the functions we encounter in analysis or algebra. Instead think of functions as *rules for computing* (by moving and plugging arguments around), a more combinatory (which does not mean combinatorial) viewpoint.

This chapter relies heavily on the masterly expositions by Barendregt [4, 5]. We also found inspiration from very informative online material by Henk Barendregt, Peter Selinger, and J.R.B. Cockett, whom we thank. Hindley and Seldin [28] and Krivine [33] are also excellent sources (and not as advanced as Barendregt [4]).

[1]Apparently, Church was fond of Greek letters.

6.1 Syntax of the Lambda-Calculus

We begin by defining the *lambda-calculus*, also called *untyped lambda-calculus* or *pure lambda-calculus*, to emphasize that the terms of this calculus are not typed. This formal system consists of

1. A set of terms, called *λ-terms*.

2. A notion of reduction, called *β-reduction*, which allows a term M to be transformed into another term N in a way that mimics a kind of evaluation.

First we define (pure) λ-terms. We have a countable set of variables $\{x_0, x_1, \ldots, x_n \ldots\}$ that correspond to the atomic λ-terms.

Definition 6.1. The *λ-terms M* are defined inductively as follows:

(1) If x_i is a variable, then x_i is a λ-term.

(2) If M and N are λ-terms, then (MN) is a λ-term called an *application*.

(3) If M is a λ-term, and x is a variable, then the expression $(\lambda x.\, M)$ is a λ-term called a *λ-abstraction*.

Note that the only difference between the λ-terms of Definition 6.1 and the raw simply-typed λ-terms of Definition 2.13 is that in Clause (3), in a λ-abstraction term $(\lambda x.\, M)$, the variable x occurs without any type information, whereas in a simply-typed λ-abstraction term $(\lambda x\colon \sigma.\, M)$, the variable x is assigned the type σ. At this stage this is only a cosmetic difference because raw λ-terms are not yet assigned types. But there are type-checking rules for assigning types to raw simply-typed λ-terms that *restrict application*, so the set of simply-typed λ-terms that type-check is much more restricted than the set of (untyped) λ-terms. In particular, no simply-typed λ-term that type-checks can be a self-application (MM). The fact that self-application is allowed in the untyped λ-calculus is what gives it its computational power (through fixed-point combinators, see Section 6.5).

Definition 6.2. The *depth* $d(M)$ of a λ-term M is defined inductively as follows.

1. If M is a variable x, then $d(x) = 0$.

2. If M is an application (M_1M_2), then $d(M) = \max\{d(M_1), d(M_2)\} + 1$.

3. If M is a λ-abstraction $(\lambda x.\, M_1)$, then $d(M) = d(M_1) + 1$.

It is pretty clear that λ-terms have representations as (ordered) labeled trees.

Definition 6.3. Given a λ-term M, the *tree* $\mathrm{tree}(M)$ representing M is defined inductively as follows:

1. If M is a variable x, then $\mathrm{tree}(M)$ is the one-node tree labeled x.

2. If M is an application (M_1M_2), then tree(M) is the tree with a binary root node labeled $.$, and with a left subtree tree(M_1) and a right subtree tree(M_2).

3. If M is a λ-abstraction $\lambda x.\, M_1$, then tree(M) is the tree with a unary root node labeled λx, and with one subtree tree(M_1).

Definition 6.3 is illustrated in Figure 6.1.

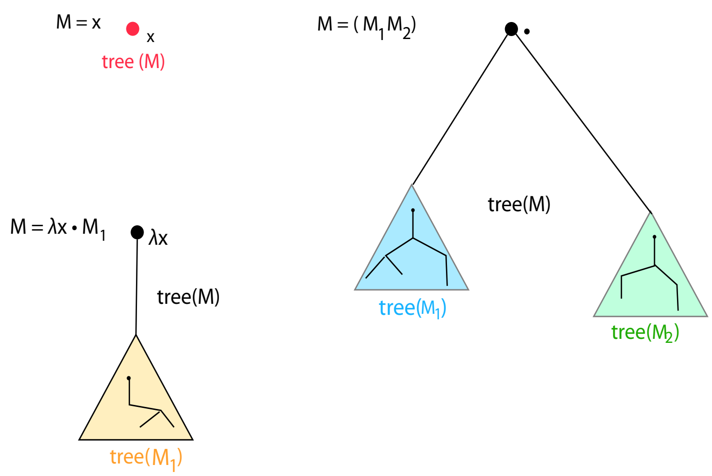


Figure 6.1: The tree tree(M) associated with a pure λ-term M.

Obviously, the depth $d(M)$ of λ-term is the depth of its tree representation tree(M).

Unfortunately λ-terms contain a profusion of parentheses so some conventions are commonly used:

(1) A term of the form

$$(\cdots((FM_1)M_2)\cdots M_n)$$

is abbreviated (association to the left) as

$$FM_1\cdots M_n.$$

(2) A term of the form

$$\lambda x_1.\,(\lambda x_2.\,(\cdots(\lambda x_n.\,M)\cdots))$$

is abbreviated (association to the right) as

$$\lambda x_1\cdots x_n.\,M.$$

Matching parentheses may be dropped or added for convenience. Here are some examples of λ-terms (and their abbreviation):

y	y
(yx)	yx
$\lambda x.\,(yx)$	$\lambda x.\,yx$
$((\lambda x.\,(yx))z)$	$(\lambda x.\,yx)z$
$(((\lambda x.\,(\lambda y.\,(yx)))z)w)$	$(\lambda xy.\,yx)zw.$

Note that $\lambda x.\,yx$ is an abbreviation for $(\lambda x.\,(yx))$, not $((\lambda x.\,y)x)$.

The variables occurring in a λ-term are free of bound.

Definition 6.4. For any λ-term M, the set $FV(M)$ of *free variables* of M and the set $BV(M)$ of *bound variables* in M are defined inductively as follows:

(1) If $M = x$ (a variable), then

$$FV(x) = \{x\}, \quad BV(x) = \emptyset.$$

(2) If $M = (M_1M_2)$, then

$$FV(M) = FV(M_1) \cup FV(M_2), \quad BV(M) = BV(M_1) \cup BV(M_2).$$

(3) if $M = (\lambda x.\,M_1)$, then

$$FV(M) = FV(M_1) - \{x\}, \quad BV(M) = BV(M_1) \cup \{x\}.$$

If $x \in FV(M_1)$, we say that the occurrences of the variable x *occur in the scope of* λ.

A λ-term M is *closed* or a *combinator* if $FV(M) = \emptyset$, that is, if it has no free variables.

For example

$$FV\big((\lambda x.\,yx)z\big) = \{y, z\}, \qquad BV\big((\lambda x.\,yx)z\big) = \{x\},$$

and

$$FV\big((\lambda xy.\,yx)zw\big) = \{z\}, \qquad BV\big((\lambda xy.\,yx)zw\big) = \{x, y\}.$$

Before proceeding with the notion of substitution we must address an issue with bound variables. The point is that bound variables are really *place-holders* so they can be renamed freely without changing the reduction behavior of the term as long as they do not clash with free variables. For example, the terms $\lambda x.\,(x(\lambda y.\,x(yx))$ and $\lambda x.\,(x(\lambda z.\,x(zx))$ should be considered as equivalent. Similarly, the terms $\lambda x.\,(x(\lambda y.\,x(yx))$ and $\lambda w.\,(w(\lambda z.\,w(zw))$ should be considered as equivalent.

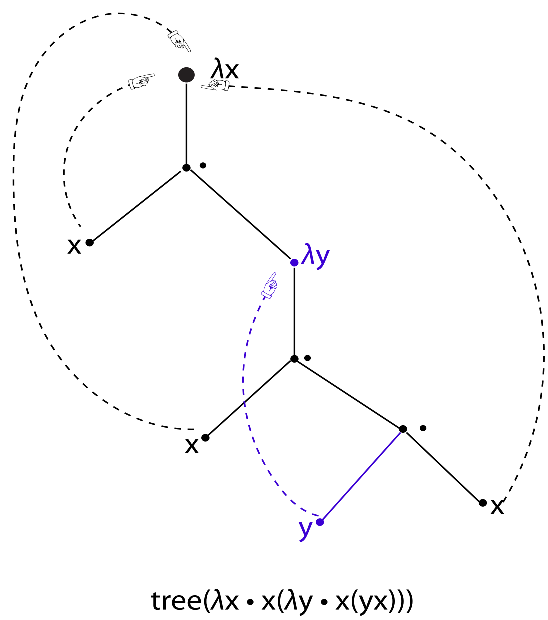


Figure 6.2: The tree representation of a λ-term with backpointers.

One way to deal with this issue is to use the tree representation of λ-terms given in Definition 6.3. For every leaf labeled with a bound variable x, we draw a backpointer to an ancestor of x determined as follows. Given a leaf labeled with a bound variable x, climb up to the closest ancestor labeled λx, and draw a backpointer to this node. Then all bound variables can be erased. An example is shown in Figure 6.2 for the term $M = \lambda x.\, x(\lambda y.\, (x(yx)))$.

A clever implementation of the idea of backpointers is the formalism of *de Bruijn indices*; see Pierce [39] (Chapter 6) or Barendregt [4] (Appendix C).

Church introduced the notion of *α-conversion* to deal with this issue. First we need to define substitutions.

A *substitution* φ is a finite set of pairs $\varphi = \{(x_1, N_1), \ldots, (x_n, N_n)\}$, where the x_i are distinct variables and the N_i are λ-terms. We write

$$\varphi = [N_1/x_1, \ldots, N_n/x_n] \quad \text{or} \quad \varphi = [x_1 := N_1, \ldots, x_n := N_n].$$

The second notation indicates more clearly that each term N_i is substituted for the variable x_i, and it seems to have been almost universally adopted.

Given a substitution $\varphi = [x_1 := N_1, \ldots, x_n := N_n]$, for any variable x_i, we denote by φ_{-x_i} the new substitution where the pair (x_i, N_i) is replaced by the pair (x_i, x_i) (that is, the new substitution leaves x_i unchanged).

Definition 6.5. Given any λ-term M and any substitution $\varphi = [x_1 := N_1, \ldots, x_n := N_n]$, we define the λ-term $M[\varphi]$, *the result of applying the substitution φ to M*, as follows:

(1) If $M = y$, with $y \neq x_i$ for $i = 1, \ldots, n$, then $M[\varphi] = y = M$.

(2) If $M = x_i$ for some $i \in \{1, \ldots, n\}$, then $M[\varphi] = N_i$.

(3) If $M = (PQ)$, then $M[\varphi] = (P[\varphi]Q[\varphi])$.

(4) If $M = \lambda x.\, N$ and $x \neq x_i$ for $i = 1, \ldots, n$, then $M[\varphi] = \lambda x.\, N[\varphi]$,

(5) If $M = \lambda x.\, N$ and $x = x_i$ for some $i \in \{1, \ldots, n\}$, then
$M[\varphi] = \lambda x.\, N[\varphi]_{-x_i}$.

The term M is *safe* for the substitution $\varphi = [x_1 := N_1, \ldots, x_n := N_n]$ if $BV(M) \cap (FV(N_1) \cup \cdots \cup FV(N_n)) = \emptyset$, that is, if the free variables in the substitution *do not* become bound.

Note that Clause (5) ensures that a substitution *only substitutes the terms N_i for the variables x_i free in M*. Thus if M is a *closed* term, then for *every* substitution φ, we have $M[\varphi] = M$.

There is a problem with the present definition of a substitution in Cases (4) and (5), which is that the result of substituting a term N_i containing the variable x free causes this variable to become bound after the substitution. We say that x is *captured*. We should only apply a substitution φ to a term M if M is safe for φ. To remedy this problem, Church defined α*-conversion*.

Definition 6.6. The binary relation $\longrightarrow_\alpha$ on λ-terms called *immediate α-conversion*[2] is the smallest relation satisfying the following properties: for all λ-terms M, N, P, Q:

$$\lambda x.\, M \longrightarrow_\alpha \lambda y.\, M[x := y], \quad \text{for all } y \notin FV(M) \cup BV(M)$$

$$\begin{array}{llll} \text{if} & M \longrightarrow_\alpha N & \text{then} & MQ \longrightarrow_\alpha NQ \quad \text{and} \quad PM \longrightarrow_\alpha PN \\ \text{if} & M \longrightarrow_\alpha N & \text{then} & \lambda x.\, M \longrightarrow_\alpha \lambda x.\, N. \end{array}$$

The least equivalence relation $\equiv_\alpha = (\longrightarrow_\alpha \cup \longrightarrow_\alpha^{-1})^*$ containing $\longrightarrow_\alpha$ (the reflexive and transitive closure of $\longrightarrow_\alpha \cup \longrightarrow_\alpha^{-1}$) is called α*-conversion*. Here $\longrightarrow_\alpha^{-1}$ denotes the converse of the relation $\longrightarrow_\alpha$, that is, $M \longrightarrow_\alpha^{-1} N$ iff $N \longrightarrow_\alpha M$.

For example,

$$\lambda fx.\, f(f(x)) = \lambda f.\, \lambda x.\, f(f(x)) \longrightarrow_\alpha \lambda f.\, \lambda y.\, f(f(y)) \longrightarrow_\alpha \lambda g.\, \lambda y.\, g(g(y)) = \lambda gy.\, g(g(y)).$$

Now given a λ-term M and a substitution $\varphi = [x_1 := N_1, \ldots, x_n := N_n]$, before applying φ to M we *first perform some α-conversion* to obtain a term $M' \equiv_\alpha M$ whose set of bound variables $BV(M')$ is disjoint from $FV(N_1) \cup \cdots \cup FV(N_n)$ so that M' *is safe for* φ, and the result of the substitution is $M'[\varphi]$. For example,

$$\big(\lambda xyz.\, (xy)z\big)(yz) \equiv_\alpha \big(\lambda xuv.\, (xu)v\big)(yz) \longrightarrow_\beta \big(\lambda uv.\, (xu)v\big)[x := yz] = \lambda uv.\, ((yz)u)v.$$

[2]We told you that Church was fond of Greek letters.

From now on, we consider two λ-terms M and M' such that $M \equiv_\alpha M'$ as identical (to be rigorous, we deal with equivalence classes of terms with respect to α-conversion). Even the experts are lax about α-conversion so we happily go along with them. The convention is that *bound variables are always renamed to avoid clashes* (with free or bound variables).

Note that the representation of λ-terms as trees with back-pointers also ensures that substitutions are safe. However, this requires some extra effort. No matter what, it takes some effort to deal properly with bound variables.

6.2 β-Reduction and β-Conversion; the Church–Rosser Theorem

The computational engine of the λ-calculus is β-reduction.

Definition 6.7. The relation $\longrightarrow_\beta$, called *immediate β-reduction*, is the smallest relation satisfying the following properties for all λ-terms M, N, P, Q:

$$(\lambda x.\, M)N \longrightarrow_\beta M[x := N], \quad \text{where } M \text{ is safe for } [x := N]$$

$$\begin{array}{llllll} \text{if} & M \longrightarrow_\beta N & \text{then} & MQ \longrightarrow_\beta NQ & \text{and} & PM \longrightarrow_\beta PN \\ \text{if} & M \longrightarrow_\beta N & \text{then} & \lambda x.\, M \longrightarrow_\beta \lambda x.\, N. & & \end{array}$$

The transitive closure of $\longrightarrow_\beta$ is denoted by $\stackrel{+}{\longrightarrow}_\beta$, the reflexive and transitive closure of $\longrightarrow_\beta$ is denoted by $\stackrel{*}{\longrightarrow}_\beta$, and we define *$\beta$-conversion*, denoted by $\stackrel{*}{\longleftrightarrow}_\beta$, as the smallest equivalence relation $\stackrel{*}{\longleftrightarrow}_\beta = (\longrightarrow_\beta \cup \longrightarrow_\beta^{-1})^*$ containing $\longrightarrow_\beta$. A subterm of the form $(\lambda x.\, M)N$ occurring in another term is called a *β-redex*. A λ-term M is a *β-normal form* if there is no λ-term N such that $M \longrightarrow_\beta N$, equivalently if M contains *no* β-redex.

For example,

$$(\lambda xy.\, x)uv = \big((\lambda x.\, (\lambda y.\, x)u\big)v \longrightarrow_\beta ((\lambda y.\, x)[x := u])v = (\lambda y.\, u)v \longrightarrow_\beta u[y := v] = u$$

and

$$((\lambda xy.\, y)uv = \big((\lambda x.\, (\lambda y.\, y)u\big)v \longrightarrow_\beta ((\lambda y.\, y)[x := u])v = (\lambda y.\, y)v \longrightarrow_\beta y[y := v] = v.$$

This shows that $\lambda xy.\, x$ behaves like the projection onto the first argument and $\lambda xy.\, y$ behaves like the projection onto the second. More interestingly, if we let $\boldsymbol{\omega} = \lambda x.\, (xx)$, then

$$\boldsymbol{\Omega} = \boldsymbol{\omega\omega} = (\lambda x.\, (xx))(\lambda x.\, (xx)) \longrightarrow_\beta (xx)[x := \lambda x.\, (xx)] = \boldsymbol{\omega\omega} = \boldsymbol{\Omega}.$$

The above example shows that β-reduction sequences may be infinite. This is a curse and a miracle of the λ-calculus!

There are even β-reductions where the evolving term grows in size:

$$\begin{aligned}(\lambda x.\, xxx)(\lambda x.\, xxx) &\xrightarrow{+}_{\beta} (\lambda x.\, xxx)(\lambda x.\, xxx)(\lambda x.\, xxx)\\ &\xrightarrow{+}_{\beta} (\lambda x.\, xxx)(\lambda x.\, xxx)(\lambda x.\, xxx)(\lambda x.\, xxx)\\ &\xrightarrow{+}_{\beta} \cdots\end{aligned}$$

In general, a λ-term contains many different β-redex. One then might wonder if there is any sort of relationship between any two terms M_1 and M_2 arising through two β-reduction sequences $M \xrightarrow{*}_{\beta} M_1$ and $M \xrightarrow{*}_{\beta} M_2$ starting with the same term M. The answer is given by the following famous theorem.

Theorem 6.1. *(Church–Rosser Theorem) The following two properties hold:*

(1) The λ-calculus is **confluent***: for any three λ-terms M, M_1, M_2, if $M \xrightarrow{*}_{\beta} M_1$ and $M \xrightarrow{*}_{\beta} M_2$, then there is some λ-term M_3 such that $M_1 \xrightarrow{*}_{\beta} M_3$ and $M_2 \xrightarrow{*}_{\beta} M_3$. See Figure 6.3.*

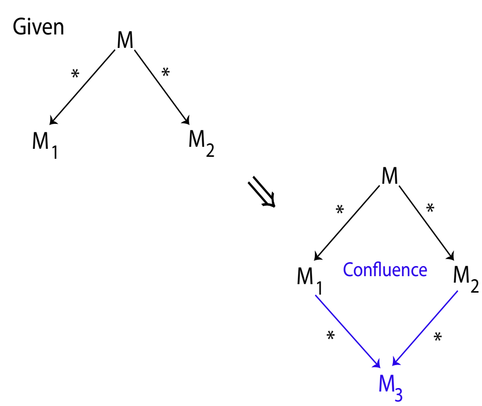


Figure 6.3: The confluence property

.

(2) The λ-calculus has the **Church–Rosser property***: for any two λ-terms M_1, M_2, if $M_1 \xleftrightarrow{*}_{\beta} M_2$, then there is some λ-term M_3 such that $M_1 \xrightarrow{*}_{\beta} M_3$ and $M_2 \xrightarrow{*}_{\beta} M_3$. See Figure 6.4.*

Furthermore (1) and (2) are equivalent, and if a λ-term M β-reduces to a β-normal form N, then N is unique (up to α-conversion).

Proof. I am not aware of any easy proof of Part (1) or Part (2) of Theorem 6.1, but the equivalence of (1) and (2) is easily shown by induction.

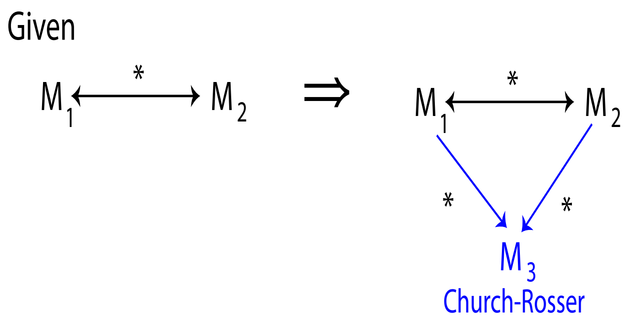


Figure 6.4: The Church–Rosser property.

Assume that (2) holds. Since $\xrightarrow{*}_{\beta}$ is contained in $\xleftrightarrow{*}_{\beta}$, if $M \xrightarrow{*}_{\beta} M_1$ and $M \xrightarrow{*}_{\beta} M_2$, then $M_1 \xleftrightarrow{*}_{\beta} M_2$, and since (2) holds, then there is some λ-term M_3 such that $M_1 \xrightarrow{*}_{\beta} M_3$ and $M_2 \xrightarrow{*}_{\beta} M_3$, which is (1).

To prove that (1) implies (2) we need the following observation.

Since $\xleftrightarrow{*}_{\beta} = (\longrightarrow_{\beta} \cup \longrightarrow_{\beta}^{-1})^*$, we see immediately that $M_1 \xleftrightarrow{*}_{\beta} M_2$ iff either

(a) $M_1 = M_2$, or

(b) there is some M_3 such that $M_1 \longrightarrow_{\beta} M_3$ and $M_3 \xleftrightarrow{*}_{\beta} M_2$, or

(c) there is some M_3 such that $M_3 \longrightarrow_{\beta} M_1$ and $M_3 \xleftrightarrow{*}_{\beta} M_2$.

Assume (1). We proceed by induction on the number of steps in $M_1 \xleftrightarrow{*}_{\beta} M_2$. If $M_1 \xleftrightarrow{*}_{\beta} M_2$, as discussed before, there are three cases.

Case a. Base case, $M_1 = M_2$. Then (2) holds with $M_3 = M_1 = M_2$.

Case b. There is some M_3 such that $M_1 \longrightarrow_{\beta} M_3$ and $M_3 \xleftrightarrow{*}_{\beta} M_2$. Since $M_3 \xleftrightarrow{*}_{\beta} M_2$ contains one less step than $M_1 \xleftrightarrow{*}_{\beta} M_2$, by the induction hypothesis there is some M_4 such that $M_3 \xrightarrow{*}_{\beta} M_4$ and $M_2 \xrightarrow{*}_{\beta} M_4$, and then $M_1 \longrightarrow_{\beta} M_3 \xrightarrow{*}_{\beta} M_4$ and $M_2 \xrightarrow{*}_{\beta} M_4$, proving (2). See Figure 6.5.

Case c. There is some M_3 such that $M_3 \longrightarrow_{\beta} M_1$ and $M_3 \xleftrightarrow{*}_{\beta} M_2$. Since $M_3 \xleftrightarrow{*}_{\beta} M_2$ contains one less step than $M_1 \xleftrightarrow{*}_{\beta} M_2$, by the induction hypothesis there is some M_4 such that $M_3 \xrightarrow{*}_{\beta} M_4$ and $M_2 \xrightarrow{*}_{\beta} M_4$. Now $M_3 \longrightarrow_{\beta} M_1$ and $M_3 \xrightarrow{*}_{\beta} M_4$, so by (1) there is some M_5 such that $M_1 \xrightarrow{*}_{\beta} M_5$ and $M_4 \xrightarrow{*}_{\beta} M_5$. Putting derivations together we get $M_1 \xrightarrow{*}_{\beta} M_5$ and $M_2 \xrightarrow{*}_{\beta} M_4 \xrightarrow{*}_{\beta} M_5$, which proves (2). See Figure 6.6.

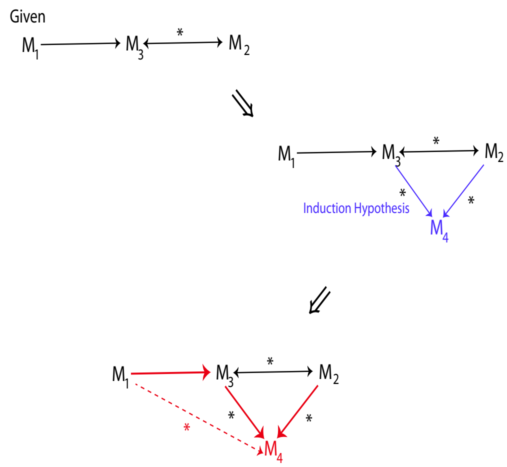


Figure 6.5: Case b.

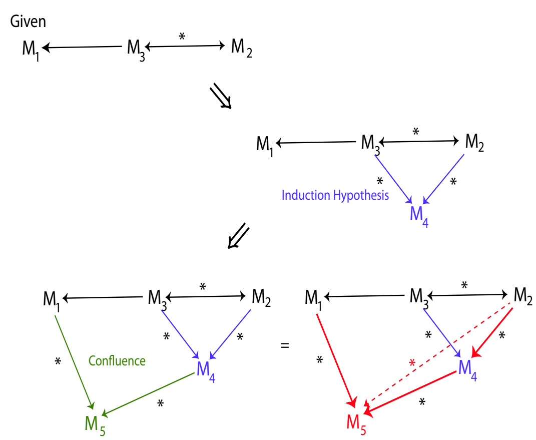


Figure 6.6: Case c.

Suppose $M \xrightarrow{*}_\beta N_1$ and $M \xrightarrow{*}_\beta N_2$ where N_1 and N_2 are both β-normal forms. Then by confluence there is some N such that $N_1 \xrightarrow{*}_\beta N$ and $N_2 \xrightarrow{*}_\beta N$. Since N_1 and N_2 are both β-normal forms, we must have $N_1 = N = N_2$ (up to α-conversion).

Barendregt gives an elegant proof of the confluence property in [4] (Chapter 11). $\square$

Another immediate corollary of the Church–Rosser theorem is that if $M \stackrel{*}{\longleftrightarrow}_\beta N$ and if N is a β-normal form, then in fact $M \xrightarrow{*}_\beta N$. We leave this fact as an exerise

This fact will be useful in showing that the recursive functions are computable in the λ-calculus.

6.3 Some Useful Combinators

In this section we provide some evidence for the expressive power of the λ-calculus.

First we make a remark about the representation of functions of several variables in the λ-calculus. The λ-calculus makes the implicit assumption that a function has a single argument. This is the idea behind application: given a term M viewed as a function and an argument N, the term (MN) represents the result of applying M to the argument N, *except that the actual evaluation is suspended*. Evaluation is performed by β-conversion. To deal with functions of several arguments we use a method known as *Currying* (after Haskell Curry). In this method, a function of n arguments is viewed as a function of one argument *taking a function of* $n-1$ *arguments as argument*. Consider the case of two arguments, the general case being similar. Consider a function $f\colon \mathbb{N}\times\mathbb{N}\to\mathbb{N}$. For any fixed x, we define the function $F_x\colon \mathbb{N}\to\mathbb{N}$ given by

$$F_x(y) = f(x,y) \qquad y\in\mathbb{N}.$$

Using the λ-notation we can write

$$F_x = \lambda y.\, f(x,y),$$

and then the function $x\mapsto F_x$, which is a function from $\mathbb{N}$ to the *set of functions* $[\mathbb{N}\to\mathbb{N}]$ (also denoted $\mathbb{N}^{\mathbb{N}}$), is denoted by the λ-term

$$F = \lambda x.\, F_x = \lambda x.\, (\lambda y.\, f(x,y)).$$

And indeed,

$$(FM)N \xrightarrow{+}_{\beta} F_M N \xrightarrow{+}_{\beta} f(M,N).$$

Remark: Currying is a way to realizing the isomorphism between the sets of functions $[\mathbb{N}\times\mathbb{N}\to\mathbb{N}]$ and $[\mathbb{N}\to[\mathbb{N}\to\mathbb{N}]]$ (or in the standard set-theoretic notation, between $\mathbb{N}^{\mathbb{N}\times\mathbb{N}}$ and $(\mathbb{N}^{\mathbb{N}})^{\mathbb{N}}$. Does this remind you of the identity

$$(m^n)^p = m^{n*p}?$$

It should.

The function space $[\mathbb{N}\to\mathbb{N}]$ is called an *exponential*. There is a very abstract way to view all this which is to say that we have an instance of a Cartesian closed category (CCC).

Proposition 6.2. *If* $\mathbf{I}, \mathbf{K}, \mathbf{K}_*$, *and* $\mathbf{S}$ *are the combinators defined by*

$$\begin{aligned}
\mathbf{I} &= \lambda x.\, x\\
\mathbf{K} &= \lambda xy.\, x\\
\mathbf{K}_* &= \lambda xy.\, y\\
\mathbf{S} &= \lambda xyz.\, (xz)(yz),
\end{aligned}$$

then for all λ-terms M, N, P, we have

$$\begin{aligned}
\mathbf{I}M &\xrightarrow{+}_\beta M \\
\mathbf{K}MN &\xrightarrow{+}_\beta M \\
\mathbf{K}_*MN &\xrightarrow{+}_\beta N \\
\mathbf{S}MNP &\xrightarrow{+}_\beta (MP)(NP) \\
\mathbf{KI} &\xrightarrow{+}_\beta \mathbf{K}_* \\
\mathbf{SKK} &\xrightarrow{+}_\beta \mathbf{I}.
\end{aligned}$$

The proof is left as an easy exercise. For example,

$$\begin{aligned}
\mathbf{S}MNP = \big(\lambda xyz.\,(xz)(yz)\big)MNP &\longrightarrow_\beta \big(\big(\lambda yz.\,(xz)(yz)\big)[x := M]\big)NP \\
&= \big(\lambda yz.\,(Mz)(yz)\big)NP \\
&\longrightarrow_\beta \big(\big(\lambda z.\,(Mz)(yz)\big)[y := N]\big)P \\
&= \big(\lambda z.\,(Mz)(Nz)\big)\big)P \\
&\longrightarrow_\beta \big((Mz)(Nz)\big)[z := P] = (MP)(NP).
\end{aligned}$$

The need for a conditional construct $\mathsf{if\ then\ else}$ such that $\mathsf{if}\ \mathbf{T}\ \mathsf{then}\ P\ \mathsf{else}\ Q$ yields P and $\mathsf{if}\ \mathbf{F}\ \mathsf{then}\ P\ \mathsf{else}\ Q$ yields Q is indispensable to write nontrivial programs. There is a trick to encode the boolean values $\mathbf{T}$ and $\mathbf{F}$ in the λ-calculus to mimick the above behavior of $\mathsf{if}\ B\ \mathsf{then}\ P\ \mathsf{else}\ Q$, provided that B is a truth value. Since everything in the λ-calculus is a function, the booleans values $\mathbf{T}$ and $\mathbf{F}$ are encoded as λ-terms. At first, this seems quite odd, but what counts is the behavior of $\mathsf{if}\ \mathbf{B}\ \mathsf{then}\ P\ \mathsf{else}\ Q$, and it works!

The truth values $\mathbf{T}, \mathbf{F}$ and the conditional construct $\mathsf{if}\ B\ \mathsf{then}\ P\ \mathsf{else}\ Q$ can be encoded in the λ-calculus as follows.

Proposition 6.3. *Consider the combinators given by $\mathbf{T} = \mathbf{K}, \mathbf{F} = \mathbf{K}_*$, and*

$$\mathsf{if\ then\ else} = \lambda bxy.\, bxy.$$

Then for all λ-terms we have

$$\begin{aligned}
\mathsf{if}\ \mathbf{T}\ \mathsf{then}\ P\ \mathsf{else}\ Q &\xrightarrow{+}_\beta P \\
\mathsf{if}\ \mathbf{F}\ \mathsf{then}\ P\ \mathsf{else}\ Q &\xrightarrow{+}_\beta Q.
\end{aligned}$$

The proof is left as an easy exercise. For example,

$$\begin{aligned}
\mathsf{if}\ \mathbf{T}\ \mathsf{then}\ P\ \mathsf{else}\ Q &= (\mathsf{if\ then\ else})\mathbf{T}PQ \\
&= \big(\lambda bxy.\, bxy\big)\mathbf{T}PQ \\
&\longrightarrow_\beta \big(\big(\lambda xy.\, bxy\big)[b := \mathbf{T}]\big)PQ = \big(\lambda xy.\, \mathbf{T}xy\big)PQ \\
&\longrightarrow_\beta \big(\big(\lambda y.\, \mathbf{T}xy\big)[x := P]\big)Q = \big(\lambda y.\, \mathbf{T}Py\big)\big)Q \\
&\longrightarrow_\beta (\mathbf{T}Py)[y := Q] = \mathbf{T}PQ \\
&= \mathbf{K}PQ \xrightarrow{+}_\beta P,
\end{aligned}$$

by Proposition 6.2.

The boolean operations $\wedge, \vee, \neg$ can be defined in terms of if then else. For example,

$$\mathsf{And}\, b_1 b_2 = \mathsf{if}\ b_1 \mathsf{then}\ (\mathsf{if}\ b_2\ \mathsf{then}\ \mathbf{T}\ \mathsf{else}\ \mathbf{F})\ \mathsf{else}\ \mathbf{F}.$$

Remark: If B is a term different from $\mathbf{T}$ or $\mathbf{F}$, then if $\mathbf{B}$ then P else Q may not reduce at all, or reduce to something different from P or Q. The problem is that the conditional statement that we designed only works properly if the input B is of the correct type, namely a boolean. If we give garbage as input, then we can't expect a correct result. The λ-calculus being type-free, it is unable to check for the validity of the input. In this sense this is a defect, but it also accounts for its power.

The ability to construct ordered pairs is also crucial.

Proposition 6.4. *For any two λ-terms M and N consider the combinator $\langle M, N\rangle$ and the combinator π_1 and π_2 given by*

$$\begin{aligned}
\langle M, N\rangle &= \lambda z.\, zMN = \lambda z.\, \mathsf{if}\ z\ \mathsf{then}\ M\ \mathsf{else}\ N \\
\pi_1 &= \lambda z.\, z\mathbf{K} \\
\pi_2 &= \lambda z.\, z\mathbf{K}_*.
\end{aligned}$$

Then

$$\begin{aligned}
\pi_1\langle M, N\rangle &\xrightarrow{+}_\beta M \\
\pi_2\langle M, N\rangle &\xrightarrow{+}_\beta N \\
\langle M, N\rangle\mathbf{T} &\xrightarrow{+}_\beta M \\
\langle M, N\rangle\mathbf{F} &\xrightarrow{+}_\beta N.
\end{aligned}$$

The proof is left as an easy exercise. For example,

$$\begin{aligned}
\pi_1\langle M, N\rangle &= \big(\lambda z.\, z\mathbf{K}\big)\big(\lambda z.\, zMN\big) \\
&\longrightarrow_\beta\ \big(z\mathbf{K}\big)[z := \lambda z.\, zMN] = \big(\lambda z.\, zMN\big)\mathbf{K} \\
&\longrightarrow_\beta\ (zMN)[z := \mathbf{K}] = \mathbf{K}MN \xrightarrow{+}_\beta M,
\end{aligned}$$

by Proposition 6.2.

In the next section we show how to encode the natural numbers in the λ-calculus and how to compute various arithmetical functions.

6.4 Representing the Natural Numbers

Historically the natural numbers were first represented in the λ-calculus by Church in the 1930's. Later in 1976 Barendregt came up with another representation which is more convenient to show that the recursive functions are λ-definable. We start with Church's representation.

First, given any two λ-terms F and M, for any natural number $n \in \mathbb{N}$, we define $F^n(M)$ inductively as follows:

$$\begin{aligned} F^0(M) &= M \\ F^{n+1}(M) &= F(F^n(M)). \end{aligned}$$

Definition 6.8. (Church Numerals) The *Church numerals* $\mathbf{c}_0, \mathbf{c}_1, \mathbf{c}_2, \ldots$ are defined by

$$\mathbf{c}_n = \lambda fx.\, f^n(x).$$

So $\mathbf{c}_0 = \lambda fx.\, x = \mathbf{K}_*$, $\mathbf{c}_1 = \lambda fx.\, fx$, $\mathbf{c}_2 = \lambda fx.\, f(fx)$, *etc.* The Church numerals are β-normal forms.

Observe that

$$\mathbf{c}_n F z = (\lambda fx.\, f^n(x))Fz \xrightarrow{+}_{\beta} F^n(z). \tag{$\dagger$}$$

This shows that $\mathbf{c}_n$ *iterates* n *times the function represented by the term* F *on initial input* z. This is the trick behind the definition of the Church numerals. This suggests the following definition.

Definition 6.9. The *iteration combinator* **Iter** is given by

$$\mathbf{Iter} = \lambda nfx.\, nfx.$$

Observe that

$$\mathbf{Iter}\, \mathbf{c}_n\, F\, X \xrightarrow{+}_{\beta} F^n X,$$

that is, the result of iterating F for n steps starting with the initial term X.

Let us show how some basic functions on the natural numbers can be defined. We begin with the constant function $\mathbf{Z}$ given by $\mathbf{Z}(n) = 0$ for all $n \in \mathbb{N}$. We claim that $\mathbf{Z_c} = \lambda x.\, \mathbf{c_0}$ works. Indeed, we have

$$\mathbf{Z_c}\, \mathbf{c}_n = (\lambda x.\, \mathbf{c_0})\mathbf{c}_n \longrightarrow_{\beta} \mathbf{c_0}[x := \mathbf{c}_n] = \mathbf{c_0}$$

since $\mathbf{c_0}$ is a closed term.

The successor function **Succ** is given by

$$\mathbf{Succ}(n) = n + 1.$$

We claim that

$$\mathbf{Succ_c} = \lambda nfx.\, f(nfx)$$

computes **Succ**. Indeed we have

$$\begin{aligned} \mathbf{Succ_c}\, \mathbf{c}_n &= (\lambda nfx.\, f(nfx))\mathbf{c}_n \\ &\longrightarrow_{\beta} \big(\lambda fx.\, f(nfx)\big)[n := \mathbf{c}_n] = \lambda fx.\, f(\mathbf{c}_n fx) \\ &\longrightarrow_{\beta} \lambda fx.\, f(f^n(x)) \\ &= \lambda fx.\, f^{n+1}(x) = \mathbf{c}_{n+1}. \end{aligned}$$

The function **IsZero** which tests whether a natural number is equal to 0 is defined by the combinator

$$\mathbf{IsZero_c} = \lambda x.\, x(\mathbf{K}\,\mathbf{F})\mathbf{T}.$$

The proof that it works is left as an exercise.

Addition and multiplication are a little more tricky to define.

Proposition 6.5. *(J.B. Rosser) Define* **Add** *and* **Mult** *as the combinators given by*

$$\begin{aligned}\mathbf{Add} &= \lambda mnfx.\, mf(nfx)\\ \mathbf{Mult} &= \lambda xyz.\, x(yz).\end{aligned}$$

We have

$$\begin{aligned}\mathbf{Add}\,\mathbf{c}_m\mathbf{c}_n &\xrightarrow{+}_\beta \mathbf{c}_{m+n}\\ \mathbf{Mult}\,\mathbf{c}_m\mathbf{c}_n &\xrightarrow{+}_\beta \mathbf{c}_{m*n}\end{aligned}$$

for all $m, n \in \mathbb{N}$.

Proof. We have

$$\begin{aligned}\mathbf{Add}\,\mathbf{c}_m\mathbf{c}_n &= (\lambda mnfx.\, mf(nfx))\mathbf{c}_m\mathbf{c}_n\\ &\xrightarrow{+}_\beta (\lambda fx.\, \mathbf{c}_m f(\mathbf{c}_n fx))\\ &\xrightarrow{+}_\beta \lambda fx.\, f^m(f^n(x))\\ &= \lambda fx.\, f^{m+n}(x) = \mathbf{c}_{m+n}.\end{aligned}$$

For multiplication we need to prove by induction on m that

$$(\mathbf{c}_n x)^m(y) \xrightarrow{*}_\beta x^{m*n}(y). \tag{$*$}$$

If $m = 0$ then both sides are equal to y.

For the induction step we have

$$\begin{aligned}(\mathbf{c}_n x)^{m+1}(y) &= \mathbf{c}_n x((\mathbf{c}_n x)^m(y))\\ &\xrightarrow{*}_\beta \mathbf{c}_n x(x^{m*n}(y)) && \text{by induction}\\ &\xrightarrow{*}_\beta x^n(x^{m*n}(y))\\ &= x^{n+m*n}(y) = x^{(m+1)*n}(y).\end{aligned}$$

We now have

$$\begin{aligned}\mathbf{Mult}\,\mathbf{c}_m\mathbf{c}_n &= (\lambda xyz.\, x(yz))\mathbf{c}_m\mathbf{c}_n\\ &\xrightarrow{+}_\beta \lambda z.\, (\mathbf{c}_m(\mathbf{c}_n z))\\ &= \lambda z.\, ((\lambda fy.\, f^m(y))(\mathbf{c}_n z))\\ &\xrightarrow{+}_\beta \lambda zy.\, (\mathbf{c}_n z)^m(y),\end{aligned}$$

and since we proved in $(*)$ that

$$(\mathbf{c}_n z)^m(y) \xrightarrow{*}_\beta z^{m*n}(y),$$

we get

$$\mathbf{Mult}\,\mathbf{c}_m\mathbf{c}_n \xrightarrow{+}_\beta \lambda zy.\,(\mathbf{c}_n z)^m(y) \xrightarrow{+}_\beta \lambda zy.\,z^{m*n}(y) = \mathbf{c}_{m*n},$$

which completes the proof. $\square$

As an exercise the reader should prove that addition and multiplication can also be defined in terms of **Iter** (see Definition 6.9) by

$$\begin{aligned}\mathbf{Add} &= \lambda mn.\,\mathbf{Iter}\,m\,\mathbf{Succ_c}\,n\\ \mathbf{Mult} &= \lambda mn.\,\mathbf{Iter}\,m\,(\mathbf{Add}\,n)\,\mathbf{c}_0.\end{aligned}$$

The above expressions are close matches to the primitive recursive definitions of addition and multiplication. To check that they work, prove that

$$\mathbf{Add}\,\mathbf{c}_m\,\mathbf{c}_n \xrightarrow{+}_\beta (\mathbf{Succ_c})^m(\mathbf{c}_n) \xrightarrow{+}_\beta \mathbf{c}_{m+n}$$

and

$$\mathbf{Mult}\,\mathbf{c}_m\,\mathbf{c}_n \xrightarrow{+}_\beta (\mathbf{Add}\,n)^m(\mathbf{c}_0) \xrightarrow{+}_\beta \mathbf{c}_{m*n}.$$

A version of the exponential function can also be defined. A function that plays an important technical role is the predecessor function **Pred** defined such that

$$\begin{aligned}\mathbf{Pred}(0) &= 0\\ \mathbf{Pred}(n+1) &= n.\end{aligned}$$

It turns out that it is quite tricky to define this function in terms of the Church numerals. Church and his students struggled for a while until Kleene found a solution in his famous 1936 paper. The story goes that Kleene found his solution when he was sittting in the dentist's chair! The trick is to make use of pairs. Kleene's solution is

$$\mathbf{Pred_K} = \lambda n.\,\pi_2(\mathbf{Iter}\,n\,\lambda z.\,\langle \mathbf{Succ_c}(\pi_1 z), \pi_1 z\rangle\,\langle \mathbf{c}_0, \mathbf{c}_0\rangle).$$

The reason this works is that we can prove that

$$(\lambda z.\,\langle \mathbf{Succ_c}(\pi_1 z), \pi_1 z\rangle)^0\langle \mathbf{c}_0, \mathbf{c}_0\rangle \xrightarrow{+}_\beta \langle \mathbf{c}_0, \mathbf{c}_0\rangle,$$

and by induction that

$$(\lambda z.\,\langle \mathbf{Succ_c}(\pi_1 z), \pi_1 z\rangle)^{n+1}\langle \mathbf{c}_0, \mathbf{c}_0\rangle \xrightarrow{+}_\beta \langle \mathbf{c}_{n+1}, \mathbf{c}_n\rangle.$$

For the base case $n = 0$ we get

$$(\lambda z.\,\langle \mathbf{Succ_c}(\pi_1 z), \pi_1 z\rangle)\langle \mathbf{c}_0, \mathbf{c}_0\rangle \xrightarrow{+}_\beta \langle \mathbf{c}_1, \mathbf{c}_0\rangle.$$

For the induction step we have

$$\begin{aligned}
&(\lambda z.\, \langle \mathbf{Succ_c}(\pi_1 z), \pi_1 z\rangle)^{n+2}\langle \mathbf{c}_0, \mathbf{c}_0\rangle = \\
&\qquad\qquad (\lambda z.\, \langle \mathbf{Succ_c}(\pi_1 z), \pi_1 z\rangle)\big((\lambda z.\, \langle \mathbf{Succ_c}(\pi_1 z), \pi_1 z\rangle)^{n+1}\langle \mathbf{c}_0, \mathbf{c}_0\rangle\big) \\
&\qquad\qquad\qquad \xrightarrow{+}_\beta (\lambda z.\, \langle \mathbf{Succ_c}(\pi_1 z), \pi_1 z\rangle)\langle \mathbf{c}_{n+1}, \mathbf{c}_n\rangle \xrightarrow{+}_\beta \langle \mathbf{c}_{n+2}, \mathbf{c}_{n+1}\rangle.
\end{aligned}$$

Here is another tricky solution due to J. Velmans (according to H. Barendregt):

$$\mathbf{Pred_c} = \lambda xyz.\, x(\lambda pq.\, q(py))(\mathbf{K}z)\mathbf{I}.$$

We leave it to the reader to verify that it works.

The ability to construct pairs together with the **Iter** combinator allows the definition of a large class of functions, because **Iter** is "type-free" in its second and third arguments so it really allows higher-order primitive recursion.

For example, the factorial function defined such that

$$\begin{aligned}
0! &= 1 \\
(n+1)! &= (n+1)n!
\end{aligned}$$

can be defined. First we define h by

$$h = \lambda xn.\, \mathbf{Mult}\, \mathbf{Succ_c} n\, x$$

and then

$$\mathbf{fact} = \lambda n.\, \pi_1(\mathbf{Iter}\, n\, \lambda z.\, \langle h(\pi_1 z)\,(\pi_2 z), \mathbf{Succ_c}(\pi_2 z)\rangle\, \langle \mathbf{c}_1, \mathbf{c}_0\rangle).$$

The above term works because

$$(\lambda z.\, \langle h(\pi_1 z)\,(\pi_2 z), \mathbf{Succ_c}(\pi_2 z)\rangle)^0\, \langle \mathbf{c}_1, \mathbf{c}_0\rangle \xrightarrow{+}_\beta \langle \mathbf{c}_1, \mathbf{c}_0\rangle = \langle \mathbf{c}_{0!}, \mathbf{c}_0\rangle,$$

and

$$(\lambda z.\, \langle h(\pi_1 z)\,(\pi_2 z), \mathbf{Succ_c}(\pi_2 z)\rangle)^{n+1}\, \langle \mathbf{c}_1, \mathbf{c}_0\rangle \xrightarrow{+}_\beta \langle \mathbf{c}_{(n+1)n!}, \mathbf{c}_{n+1}\rangle = \langle \mathbf{c}_{(n+1)!}, \mathbf{c}_{n+1}\rangle.$$

We leave the details as an exercise.

Barendregt came up with another way of representing the natural numbers that makes things easier.

Definition 6.10. (Barendregt Numerals) The *Barendregt numerals* $\mathbf{b}_n$ are defined as follows:

$$\begin{aligned}
\mathbf{b}_0 &= \mathbf{I} = \lambda x.\, x \\
\mathbf{b}_{n+1} &= \langle \mathbf{F}, \mathbf{b}_n\rangle.
\end{aligned}$$

The Barendregt numerals are β-normal forms. Barendregt uses the notation $\ulcorner n \urcorner$ instead of $\mathbf{b}_n$ but this notation is also used for the Church numerals by other authors so we prefer using $\mathbf{b}_n$ (which is consistent with the use of $\mathbf{c}_n$ for the Church numerals). The Barendregt numerals are tuples, which makes operating on them simpler than the Church numerals which encode n as the composition f^n.

Proposition 6.6. *The functions* **Succ**, **Pred** *and* **IsZero** *are defined in terms of the Barendregt numerals by the combinators*

$$\begin{aligned}\mathbf{Succ_b} &= \lambda x.\, \langle \mathbf{F}, x\rangle \\ \mathbf{Pred_b} &= \lambda x.\, (x\mathbf{F}) \\ \mathbf{IsZero_b} &= \lambda x.\, (x\mathbf{T}),\end{aligned}$$

and we have

$$\begin{aligned}\mathbf{Succ_b}\, \mathbf{b}_n &\xrightarrow{+}_\beta \mathbf{b}_{n+1} \\ \mathbf{Pred_b}\, \mathbf{b}_0 &\xrightarrow{+}_\beta \mathbf{b}_0 \\ \mathbf{Pred_b}\, \mathbf{b}_{n+1} &\xrightarrow{+}_\beta \mathbf{b}_n \\ \mathbf{IsZero_b}\, \mathbf{b}_0 &\xrightarrow{+}_\beta \mathbf{T} \\ \mathbf{IsZero_b}\, \mathbf{b}_{n+1} &\xrightarrow{+}_\beta \mathbf{F}.\end{aligned}$$

The proof is left as an exercise.

Since there is an obvious bijection between the Church combinators and the Barendregt combinators there should be combinators effecting the translations. Indeed we have the following result.

Proposition 6.7. *The combinator* T *given by*

$$T = \lambda x.\, (x\mathbf{Succ_b})\mathbf{b}_0$$

has the property that

$$T\, \mathbf{c}_n \xrightarrow{+}_\beta \mathbf{b}_n \qquad \textit{for all } n \in \mathbb{N}.$$

Proof. We proceed by induction on n. For the base case

$$\begin{aligned}T\, \mathbf{c}_0 &= (\lambda x.\, (x\mathbf{Succ_b})\mathbf{b}_0)\mathbf{c}_0 \\ &\xrightarrow{+}_\beta \mathbf{c}_0(\mathbf{Succ_b})\mathbf{b}_0 \\ &\xrightarrow{+}_\beta \mathbf{b}_0.\end{aligned}$$

For the induction step,

$$\begin{aligned}T\, \mathbf{c}_n &= (\lambda x.\, (x\mathbf{Succ_b})\mathbf{b}_0)\mathbf{c}_n \\ &\xrightarrow{+}_\beta (\mathbf{c}_n\mathbf{Succ_b})\mathbf{b}_0 \\ &\xrightarrow{+}_\beta \mathbf{Succ_b}^n(\mathbf{b}_0).\end{aligned}$$

Thus we need to prove that

$$\mathbf{Succ_b}^n(\mathbf{b}_0) \xrightarrow{+}_\beta \mathbf{b}_n. \tag{$*$}$$

For the base case $n = 0$, the left-hand side reduces to $\mathbf{b}_0$.

For the induction step, we have

$$\begin{aligned}
\mathbf{Succ_b}^{n+1}(\mathbf{b}_0) &= \mathbf{Succ_b}(\mathbf{Succ_b}^{n}(\mathbf{b}_0)) \\
&= \xrightarrow{+}_\beta \mathbf{Succ_b}(\mathbf{b}_n) && \text{by induction} \\
&= \xrightarrow{+}_\beta \mathbf{b}_{n+1},
\end{aligned}$$

which conclude the proof. $\square$

There is also a combinator defining the inverse map but it is defined recursively and we don't know how to express recursive definitions in the λ-calculus. This is achieved by using fixed-point combinators.

6.5 Fixed-Point Combinators and Recursively Defined Functions

Fixed-point combinators are the key to the definability of recursive functions in the λ-calculus. We begin with the $\mathbf{Y}$-combinator due to Curry.

Proposition 6.8. *(Curry $\mathbf{Y}$-combinator) If we define the combinator $\mathbf{Y}$ as*

$$\mathbf{Y} = \lambda f.\, (\lambda x.\, f(xx))(\lambda x.\, f(xx)),$$

then for any λ-term F we have

$$F(\mathbf{Y}F) \overset{*}{\longleftrightarrow}_\beta \mathbf{Y}F.$$

Proof. Write $W = \lambda x.\, F(xx)$. We have

$$F(\mathbf{Y}F) = F\Big(\big(\lambda f.\, (\lambda x.\, f(xx))(\lambda x.\, f(xx))\big)F\Big) \longrightarrow_\beta F\big((\lambda x.\, F(xx))(\lambda x.\, F(xx))\big) = F(WW),$$

and

$$\begin{aligned}
\mathbf{Y}F = \big(\lambda f.\, (\lambda x.\, f(xx))(\lambda x.\, f(xx))\big)F &\longrightarrow_\beta (\lambda x.\, F(xx))(\lambda x.\, F(xx)) = (\lambda x.\, F(xx))W \\
&\longrightarrow_\beta F(WW).
\end{aligned}$$

Therefore $F(\mathbf{Y}F) \overset{*}{\longleftrightarrow}_\beta \mathbf{Y}F$, as claimed. $\square$

Observe that neither $F(\mathbf{Y}F) \xrightarrow{+}_\beta \mathbf{Y}F$ nor $\mathbf{Y}F \xrightarrow{+}_\beta F(\mathbf{Y}F)$. This is a slight disadvantage of the Curry $\mathbf{Y}$-combinator. Turing came up with another fixed-point combinator that does not have this problem.

Proposition 6.9. *(Turing $\mathbf{\Theta}$-combinator) If we define the combinator $\mathbf{\Theta}$ as*

$$\mathbf{\Theta} = (\lambda xy.\, y(xxy))(\lambda xy.\, y(xxy)),$$

then for any λ-term F we have

$$\mathbf{\Theta} F \xrightarrow{+}_{\beta} F(\mathbf{\Theta} F).$$

Proof. If we write $A = (\lambda xy.\, y(xxy))$, then $\mathbf{\Theta} = AA$. We have

$$\begin{aligned} \mathbf{\Theta} F &= (AA)F = ((\lambda xy.\, y(xxy))A)F \\ &\longrightarrow_{\beta} (\lambda y.\, y(AAy))F \\ &\longrightarrow_{\beta} F(AAF) \\ &= F(\mathbf{\Theta} F), \end{aligned}$$

as claimed. ☐

Now we show how to use the fixed-point combinators to represent recursively-defined functions in the λ-calculus. For example, there is a combinator G such that

$$GX \xrightarrow{+}_{\beta} X(XG) \quad \text{for all } X.$$

Informally the idea is to consider the "functional" $F = \lambda gx.\, x(xg)$, and to find a fixed-point of this functional. Pick

$$G = \mathbf{\Theta} \lambda gx.\, x(xg) = \mathbf{\Theta} F.$$

Since by Proposition 6.9 we have $G = \mathbf{\Theta} F \xrightarrow{+}_{\beta} F(\mathbf{\Theta} F) = FG$, and we also have

$$FG = (\lambda gx.\, x(xg))G \longrightarrow_{\beta} \lambda x.\, x(xG),$$

so $G \xrightarrow{+}_{\beta} \lambda x.\, x(xG)$, which implies

$$GX \xrightarrow{+}_{\beta} (\lambda x.\, x(xG))X \longrightarrow_{\beta} X(XG).$$

In general, if we want to define a function G recursively such that

$$GX \xrightarrow{+}_{\beta} M(X, G)$$

where $M(X, G)$ is λ-term containing recursive occurrences of G, we let $F = \lambda gx.\, M(x, g)$ and

$$G = \mathbf{\Theta} F.$$

Then we have

$$G \xrightarrow{+}_{\beta} FG = (\lambda gx.\, M(x, g))G \longrightarrow_{\beta} \lambda x.\, M(x, g)[g := G] = \lambda x.\, M(x, G),$$

so

$$GX \xrightarrow{+}_{\beta} (\lambda x.\, M(x, G))X \longrightarrow_{\beta} M(x, G)[x := X] = M(X, G),$$

as desired.

As another example, here is how the factorial function can be defined (using the Church numerals). Let

$$F = \lambda gn.\,\mathsf{if}\ \mathbf{IsZero_c}\,n\ \mathsf{then}\ \mathbf{c}_1\ \mathsf{else}\ \mathbf{Mult}\,n\,g(\mathbf{Pred_c}\,n).$$

Then the term $G = \mathbf{\Theta}F$ defines factorial function. The verification of the above fact is left as an exercise.

As usual with recursive definitions there is no guarantee that the function that we obtain terminates for all input. For example, if we consider

$$F = \lambda gn.\,\mathsf{if}\ \mathbf{IsZero_c}\,n\ \mathsf{then}\ \mathbf{c}_1\ \mathsf{else}\ \mathbf{Mult}\,n\,g(\mathbf{Succ_c}\,n)$$

then for $n \geq 1$ the reduction behavior is

$$G\mathbf{c}_n \xrightarrow{+}_\beta \mathbf{Mult}\,\mathbf{c}_n\,G\,\mathbf{c}_{n+1},$$

which does not terminate.

We leave it as an exercise to show that the inverse of the function T mapping the Church numerals to the Barendregt numerals is given by the combinator

$$T^{-1} = \mathbf{\Theta}(\lambda fx.\,\mathsf{if}\ \mathbf{IsZero_b}\,x\ \mathsf{then}\ \mathbf{c}_0\ \mathbf{else}\ \mathbf{Succ_c}(f(\mathbf{Pred_b}\,x))).$$

It is remarkable that the λ-calculus allows the implementation of *arbitrary recursion* without a stack, just using λ-terms are the data-structure and β-reduction. This does not mean that this evaluation mechanism is efficient but this is another story (as well as evaluation strategies, which have to do with parameter-passing strategies, call-by-name, call-by-value).

Now we have all the ingredients to show that all the (total) computable functions are definable in the λ-calculus.

6.6 λ-Definability of the Computable Functions

Let us begin by reviewing the definition of the computable functions (recursive functions) (à la Herbrand–Gödel–Kleene). For our purposes it suffices to consider functions (partial or total) $f\colon \mathbb{N}^n \to \mathbb{N}$ as opposed to the more general case of functions $f\colon (\Sigma^*)^n \to \Sigma^*$ defined on strings.

Definition 6.11. The *base functions* are the functions Z, S, P_i^n defined as follows:

(1) The constant *zero function* Z such that

$$Z(n) = 0, \qquad \text{for all } n \in \mathbb{N}.$$

(2) The *successor function* S such that

$$S(n) = n + 1, \qquad \text{for all } n \in \mathbb{N}.$$

(3) For every $n \geq 1$ and every i with $1 \leq i \leq n$, the *projection function* P_i^n such that

$$P_i^n(x_1, \ldots, x_n) = x_i, \qquad x_1, \ldots, x_n \in \mathbb{N}.$$

Next comes (extended) composition.

Definition 6.12. Given any partial or total function$g\colon \mathbb{N}^m \to \mathbb{N}$ ($m \geq 1$) and any m partial or total functions $h_i\colon \mathbb{N}^n \to \mathbb{N}$ ($n \geq 1$), the *composition of g and $h_1, \ldots, h_m$*, denoted $g \circ (h_1, \ldots, h_m)$, is the partial or total function function $f\colon \mathbb{N}^n \to \mathbb{N}$ given by

$$f(x_1, \ldots, x_n) = g(h_1(x_1, \ldots, x_n), \ldots, h_m(x_1, \ldots, x_n)), \qquad x_1, \ldots, x_n \in \mathbb{N}.$$

If g or any of the h_i are partial functions, then $f(x_1, \ldots, x_n)$ is defined if and only if *all $h_i(x_1, \ldots, x_n)$ are defined and $g(h_1(x_1, \ldots, x_n), \ldots, h_m(x_1, \ldots, x_n))$ is defined.*

Note that even if g "ignores" one of its arguments, say the ith one, $g(h_1(x_1, \ldots, x_n), \ldots, h_m(x_1, \ldots, x_n))$ is undefined if $h_i(x_1, \ldots, x_n)$ is undefined.

Definition 6.13. Given any partial or total functions $g\colon \mathbb{N}^m \to \mathbb{N}$ and $h\colon \mathbb{N}^{m+2} \to \mathbb{N}$ ($m \geq 1$), the partial or total function function $f\colon \mathbb{N}^{m+1} \to \mathbb{N}$ is defined by *primitive recursion* from g and h if f is given by:

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

for all $n, x_1, \ldots, x_m \in \mathbb{N}$. If $m = 0$, then g is some fixed natural number and we have

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

It can be shown that if g and h are total functions, then so if f.

Note that the second clause of the definition of primitive recursion is

$$f(n+1, x_1, \ldots, x_m) = h(f(n, x_1, \ldots, x_m), n, x_1, \ldots, x_m) \tag{$*_1$}$$

but in an earlier definition it was

$$f(n+1, x_1, \ldots, x_m) = h(n, f(n, x_1, \ldots, x_m), x_1, \ldots, x_m), \tag{$*_2$}$$

with the first two arguments of h permuted. Since

$$\begin{aligned} h \circ (P_2^{m+2}, P_1^{m+2}, P_3^{m+2}, \ldots, P_{m+2}^{m+2})&(n, f(n, x_1, \ldots, x_m), x_1, \ldots, x_m) \\ &= h(f(n, x_1, \ldots, x_m), n, x_1, \ldots, x_m) \end{aligned}$$

and

$$h \circ (P_2^{m+2}, P_1^{m+2}, P_3^{m+2}, \ldots, P_{m+2}^{m+2})(f(n, x_1, \ldots, x_m), n, x_1, \ldots, x_m)$$
$$= h(n, f(n, x_1, \ldots, x_m), x_1, \ldots, x_m),$$

the two definitions are equivalent. In this section we chose version $(*_1)$ because it matches the treatment in Barendregt [4] and will make it easier for the reader to follow Barendregt [4] if they wish.

The last operation is *minimization* (sometimes called minimalization).

Definition 6.14. Given any partial or total function $g \colon \mathbb{N}^{m+1} \to \mathbb{N}$ ($m \geq 0$), the partial or total function function $f \colon \mathbb{N}^m \to \mathbb{N}$ is defined as follows: for all $x_1, \ldots, x_m \in \mathbb{N}$,

$$f(x_1, \ldots, x_m) = \text{the least } n \in \mathbb{N} \text{ such that } g(n, x_1, \ldots, x_m) = 0,$$

and undefined if there is no n such that $g(n, x_1, \ldots, x_m) = 0$. We say that f is *defined by minimization from* g, and we write

$$f(x_1, \ldots, x_m) = \mu x[g(x, x_1, \ldots, x_m) = 0].$$

For short, we write $f = \mu g$.

Even if g is a total function, f may be undefined for some (or all) of its inputs.

Definition 6.15. (Herbrand–Gödel–Kleene) The set of *partial computable* (or *partial recursive*) functions is the smallest set of partial functions (defined on $\mathbb{N}^n$ for some $n \geq 1$) which contains the base functions and is closed under

(1) Composition.

(2) Primitive recursion.

(3) Minimization.

The set of *computable* (or *recursive*) functions is the subset of partial computable functions that are total functions (that is, defined for all input).

We proved earlier the Kleene normal form, which says that *every* partial computable function $f \colon \mathbb{N}^m \to \mathbb{N}$ is computable as

$$f = g \circ \mu h,$$

for some *primitive recursive functions* $g \colon \mathbb{N} \to \mathbb{N}$ and $h \colon \mathbb{N}^{m+1} \to \mathbb{N}$. The significance of this result is that f is built up from *total functions* using composition and primitive recursion, and only a single minimization is needed at the end.

Before stating our main theorem, we need to define what it means for a (numerical) function in the λ-calculus. This requires some care to handle partial functions.

Since there are combinators for translating Church numerals to Barendregt numerals and vice-versa, it does not matter which numerals we pick. We pick the Church numerals because primitive recursion is definable without using a fixed-point combinator.

Definition 6.16. A function (partial or total) $f\colon \mathbb{N}^n \to \mathbb{N}$ is *λ-definable* if for all $m_1, \ldots,$ $m_n \in \mathbb{N}$, there is a combinator (a closed λ-term) F with the following properties:

(1) The value $f(m_1, \ldots, m_n)$ is defined if and only if $F\mathbf{c}_{m_1} \cdots \mathbf{c}_{m_n}$ reduces to a β-normal form (necessarily unique by the Church–Rosser theorem).

(2) If $f(m_1, \ldots, m_n)$ is defined, then

$$F\mathbf{c}_{m_1} \cdots \mathbf{c}_{m_n} \stackrel{*}{\longleftrightarrow}_\beta \mathbf{c}_{f(m_1,\ldots,m_n)}.$$

In view of the Church–Rosser theorem (Theorem 6.1) and the fact that $\mathbf{c}_{f(m_1,\ldots,m_n)}$ is a β-normal form, we can replace

$$F\mathbf{c}_{m_1} \cdots \mathbf{c}_{m_n} \stackrel{*}{\longleftrightarrow}_\beta \mathbf{c}_{f(m_1,\ldots,m_n)}$$

by

$$F\mathbf{c}_{m_1} \cdots \mathbf{c}_{m_n} \stackrel{*}{\longrightarrow}_\beta \mathbf{c}_{f(m_1,\ldots,m_n)}$$

Note that the termination behavior of f on inputs $m_1, \ldots, m_n$ has to *match* the reduction behavior of $F\mathbf{c}_{m_1} \cdots \mathbf{c}_{m_n}$, namely $f(m_1, \ldots, m_n)$ is undefined if *no* reduction sequence from $F\mathbf{c}_{m_1} \cdots \mathbf{c}_{m_n}$ reaches a β-normal form. Condition (2) ensures that if $f(m_1, \ldots, m_n)$ is defined, then the correct value $\mathbf{c}_{f(m_1,\ldots,m_n)}$ is computed by some reduction sequence from $F\mathbf{c}_{m_1} \cdots \mathbf{c}_{m_n}$. If we only care about total functions then we require that $F\mathbf{c}_{m_1} \cdots \mathbf{c}_{m_n}$ reduces to a β-normal for *all* $m_1, \ldots, m_n$ and (2). A stronger and more elegant version of λ-definabilty that better captures when a function is undefined for some input is considered in Section 6.7 .

We have the following remarkable theorems.

Theorem 6.10. *If a total function $f\colon \mathbb{N}^n \to \mathbb{N}$ is λ-definable, then it is (total) computable. If a partial function $f\colon \mathbb{N}^n \to \mathbb{N}$ is λ-definable, then it is partial computable.*

Although Theorem 6.10 is intuitively obvious since computation by β-reduction sequences are "clearly" computable, a detailed proof is long and very tedious. One has to define primitive recursive functions to mimick β-conversion, *etc*. Most books sweep this issue under the rug. Barendregt observes that the "λ-calculus is recursively axiomatized," which implies that the graph of the function beeing defined is recursively enumerable, but no details are provided; see Barendregt [4] (Chapter 6, Theorem 6.3.13). Kleene (1936) provides a detailed and very tedious proof. This is an amazing paper, but very hard to read. If the reader is not content she/he should work out the details over many long lonely evenings.

Theorem 6.11. *(Kleene, 1936) If a (total) function $f\colon \mathbb{N}^n \to \mathbb{N}$ is computable, then it is λ-definable. If a (partial) function $f\colon \mathbb{N}^n \to \mathbb{N}$ is is partial computable, then it is λ-definable.*

Proof. First we assume all functions to be total. There are several steps.

Step 1. The base functions are λ-definable.

We already showed that $\mathbf{Z_c}$ computes Z and that $\mathbf{Succ_c}$ computes S. Observe that $\mathbf{U}_i^n$ given by

$$\mathbf{U}_i^n = \lambda x_1 \cdots x_n.\, x_i$$

computes P_i^n.

Step 2. Closure under composition.

If g is λ-defined by the combinator G and $h_1, \ldots, h_m$ are λ-defined by the combinators $H_1, \ldots, H_m$, then $g \circ (h_1, \ldots, h_m)$ is λ-defined by

$$F = \lambda x_1 \cdots x_n.\, G(H_1 x_1 \cdots x_n) \ldots (H_m x_1 \cdots x_n).$$

Since the functions are total, there is no problem.

Step 3. Closure under primitive recursion.

We could use a fixed-point combinator but the combinator **Iter** and pairing do the job. If f is defined by primitive recursion from g and h, and if G λ-defines g and H λ-defines h, then f is λ-defined by

$$F = \lambda n x_1 \cdots x_m.\, \pi_1\big(\mathbf{Iter}\, n\, \lambda z.\, \langle H\, \pi_1 z\, \pi_2 z\, x_1 \cdots x_m,\, \mathbf{Succ_c}(\pi_2 z)\rangle\, \langle G x_1 \cdots x_m, \mathbf{c}_0\rangle\big).$$

The reason F works is that we can prove by induction that

$$\big(\lambda z.\, \langle H\, \pi_1 z\, \pi_2 z\, \mathbf{c}_{n_1} \cdots \mathbf{c}_{n_m},\, \mathbf{Succ_c}(\pi_2 z)\rangle\big)^n \langle G \mathbf{c}_{n_1} \cdots \mathbf{c}_{n_m}, \mathbf{c}_0\rangle \xrightarrow{+}_\beta \langle \mathbf{c}_{f(n,n_1,\ldots,n_m)}, \mathbf{c}_n\rangle.$$

For the base case $n = 0$,

$$\begin{aligned}&\big(\lambda z.\, \langle H\, \pi_1 z\, \pi_2 z\, \mathbf{c}_{n_1} \cdots \mathbf{c}_{n_m},\, \mathbf{Succ_c}(\pi_2 z)\rangle\big)^0 \langle G \mathbf{c}_{n_1} \cdots \mathbf{c}_{n_m}, \mathbf{c}_0\rangle \\ &\qquad \xrightarrow{+}_\beta \langle G \mathbf{c}_{n_1} \cdots \mathbf{c}_{n_m}, \mathbf{c}_0\rangle = \langle \mathbf{c}_{g(n_1,\ldots,n_m)}, \mathbf{c}_0\rangle = \langle \mathbf{c}_{f(0,n_1,\ldots,n_m)}, \mathbf{c}_0\rangle.\end{aligned}$$

For the induction step,

$$\begin{aligned}&\big(\lambda z.\, \langle H\, \pi_1 z\, \pi_2 z\, \mathbf{c}_{n_1} \cdots \mathbf{c}_{n_m},\, \mathbf{Succ_c}(\pi_2 z)\rangle\big)^{n+1} \langle G \mathbf{c}_{n_1} \cdots \mathbf{c}_{n_m}, \mathbf{c}_0\rangle \\ &\qquad = \big(\lambda z.\, \langle H\, \pi_1 z\, \pi_2 z\, \mathbf{c}_{n_1} \cdots \mathbf{c}_{n_m},\, \mathbf{Succ_c}(\pi_2 z)\rangle\big) \\ &\qquad\quad \Big(\big(\lambda z.\, \langle H\, \pi_1 z\, \pi_2 z\, \mathbf{c}_{n_1} \cdots \mathbf{c}_{n_m},\, \mathbf{Succ_c}(\pi_2 z)\rangle\big)^n \langle G \mathbf{c}_{n_1} \cdots \mathbf{c}_{n_m}, \mathbf{c}_0\rangle\Big) \\ &\qquad \xrightarrow{+}_\beta \big(\lambda z.\, \langle H\, \pi_1 z\, \pi_2 z\, \mathbf{c}_{n_1} \cdots \mathbf{c}_{n_m},\, \mathbf{Succ_c}(\pi_2 z)\rangle\big) \langle \mathbf{c}_{f(n,n_1,\ldots,n_m)}, \mathbf{c}_n\rangle \\ &\qquad \xrightarrow{+}_\beta \langle H \mathbf{c}_{f(n,n_1,\ldots,n_m)}\, \mathbf{c}_n\, \mathbf{c}_{n_1} \cdots \mathbf{c}_{n_m},\, \mathbf{Succ_c}\, \mathbf{c}_n\rangle \\ &\qquad \xrightarrow{+}_\beta \langle \mathbf{c}_{h(f(n,n_1,\ldots,n_m),n,n_1,\ldots,n_m)}, \mathbf{c}_{n+1}\rangle = \langle \mathbf{c}_{f(n+1,n_1,\ldots,n_m)}, \mathbf{c}_{n+1}\rangle.\end{aligned}$$

Since the functions are total, there is no problem.

We can also show that primitive recursion can be achieved using a fixed-point combinator. Define the combinators J and F by

$$J = \lambda f x x_1 \cdots x_m.\, \mathsf{if}\, \mathbf{IsZero_c} x\, \mathsf{then}\, G x_1 \cdots x_m\, \mathsf{else}\, H(f(\mathbf{Pred_c} x)\, x_1 \cdots x_m)(\mathbf{Pred_c}\, x)\, x_1 \cdots x_m,$$

and

$$F = \mathbf{\Theta} J.$$

Then F λ-defines f, and since the functions are total, there is no problem. This method must be used if we use the Barendregt numerals.

Step 4. Closure under minimization.

Suppose f is total and defined by minimization from g and that g is λ-defined by G.

Define the combinators J and F by

$$J = \lambda f x x_1 \cdots x_m.\, \mathsf{if}\ \mathbf{IsZero_c}\, Gx\, x_1 \cdots x_m\ \mathsf{then}\ x\ \mathsf{else}\ f(\mathbf{Succ_c}\, x)\, x_1 \cdots x_m$$

and

$$F = \mathbf{\Theta} J.$$

It is not hard to check that

$$F\, \mathbf{c}_n\, \mathbf{c}_{n_1} \ldots \mathbf{c}_{n_n} \xrightarrow{+}_\beta \begin{cases} \mathbf{c}_n & \text{if } g(n, n_1, \ldots, n_m) = 0 \\ F\, \mathbf{c}_{n+1}\, \mathbf{c}_{n_1} \cdots \mathbf{c}_{n_n} & \text{otherwise,} \end{cases}$$

and we can use this to prove that F λ-defines f. Since we assumed that f is total, some least n will be found. We leave the details as an exercise.

This finishes the proof that every total computable function is λ-definable.

To prove the result for the partial computable functions we appeal to the Kleene normal form: every partial computable function $f\colon \mathbb{N}^m \to \mathbb{N}$ is computable as

$$f = g \circ \mu h,$$

for some *primitive recursive functions* $g\colon \mathbb{N} \to \mathbb{N}$ and $h\colon \mathbb{N}^{m+1} \to \mathbb{N}$. Then our previous proof yields combinators G and H that λ-define g and h. The minimization of h may fail but since g is a total function of a single argument, $f(n_1, \ldots, n_m)$ is defined iff $g(\mu_n[h(n, n_1, \ldots, n_m) = 0])$ is defined so it should be clear that F computes f, but the reader may want to provide a rigorous argument. A detailed proof is given in Hindley and Seldin [28] (Chapter 4, Theorem 4.18). $\square$

Combining Theorem 6.10 and Theorem 6.11 we have established the remarkable result that the set of λ-definable total functions is exactly the set of (total) computable functions, and similarly for partial functions. So the λ-calculus has universal computing power.

Remark: With some work, it is possible to show that lists can be represented in the λ-calculus. Since a Turing machine tape can be viewed as a list, it should be possible (but very tedious) to simulate a Turing machine in the λ-calculus. This simulation should be somewhat analogous to the proof that a Turing machine computes a computable function (defined à la Herbrand–Gödel–Kleene).

Since the λ-calculus has the same power as Turing machines we should expect some undecidabity results analogous to the undecidability of the halting problem or Rice's theorem. We state the following analog of Rice's theorem without proof. It is a corollary of a theorem known as the Scott–Curry theorem.

Theorem 6.12. *(D. Scott) Let $\mathcal{A}$ be any nonempty set of λ-terms not equal to the set of all λ-terms. If $\mathcal{A}$ is closed under β-reduction, then it is not computable (not recursive).*

Theorem 6.12 is proven in Barendregt [4] (Chapter 6, Theorem 6.6.2) and Barendregt [5].

As a corollary of Theorem 6.12 it is undecidable whether a λ-term has a β-normal form, a result originally proved by Church. This is an analog of the undecidability of the halting problem, but it seems more spectacular because the syntax of λ-terms is really very simple. The problem is that β-reduction is very powerful and elusive.

6.7 Definability of Functions in Typed Lambda-Calculi

In the pure λ-calculus, some λ-terms have no β-normal form, and worse, it is undecidable whether a λ-term has a β-normal form. In contrast, by Theorem 2.12, *every* raw λ-term that type-checks in the simply-typed λ-calculus has a β-normal form. Thus it is natural to ask whether the natural numbers are definable in the simply-typed λ-calculus because if the answer is positive, then the numerical functions definable in the simply-typed λ-calculus are guaranteed to be total.

This indeed possible. If we pick any base type σ, then we can define typed Church numerals $\mathbf{c}_n$ as terms of type $\mathsf{Nat}_\sigma = (\sigma \to \sigma) \to (\sigma \to \sigma)$, by

$$\mathbf{c}_n = \lambda f \colon (\sigma \to \sigma).\, \lambda x \colon \sigma.\, f^n(x).$$

The notion of λ-definable function is defined just as before. Then we can define **Add** and **Mult** as terms of type $\mathsf{Nat}_\sigma \to (\mathsf{Nat}_\sigma \to \mathsf{Nat}_\sigma)$ essentially as before, but surprise, not much more is definable. Among other things, strong typing of terms restricts the iterator combinator too much. It was shown by Schwichtenberg and Statman that the numerical functions definable in the simply-typed λ-calculus are the extended polynomials; see Statman [47] and Troelstra and Schwichtenberg [50]. The extended polynomials are the smallest class of numerical functions closed under composition containing

1. The constant functions 0 and 1.

2. The projections.

3. Addition and multiplication.

4. The function $\mathbf{IsZero_c}$.

Is there a way to get a larger class of total functions?

There are indeed various ways of doing this. One method is to add the natural numbers and the booleans as data types to the simply-typed λ-calculus, and to also add product types, an iterator combinator, and some new reduction rules. This way we obtain a system equivalent to Gödel's system T. A large class of numerical total functions containing the primitive recursive functions is definable in this system; see Girard–Lafond–Taylor [21]. Although theoretically interesting, this is not a practical system.

Another wilder method is to allow more general types to the simply-typed λ-calculus, the so-called *second-order types* or *polymorphic types*. In addition to base types, we allow *type variables* (often denoted $X, Y, \ldots$) ranging over simple types and new types of the form $\forall X.\, \sigma$.[3] For example, $\forall X.\, (X \to X)$ is such a new type, and so is

$$\forall X.\, (X \to ((X \to X) \to X)).$$

Actually, the second-order types that we just defined are special cases of the QBF (quantified boolean formulae) arising in complexity theory restricted to implication and universal quantifiers; see Section 13.3. Remarkably, the other connectives $\wedge, \vee, \neg$ and $\exists$ are definable in terms of $\to$ (as a logical connective, $\Rightarrow$) and $\forall$; see Troelstra and Schwichtenberg [50] (Chapter 11).

Remark: The type

$$\mathsf{Nat} = \forall X.\, (X \to ((X \to X) \to X)).$$

can be chosen to represent the type of the natural numbers. The type of the natural numbers can also be chosen to be

$$\forall X.\, (X \to X) \to (X \to X).$$

This makes essentially no difference but the first choice has some technical advantages.

There is also a new form of *type abstraction*, $\Lambda X.\, M$, and of *type application*, $M\sigma$, where M is a λ-term and σ is a type. There are two new typing rules:

$$\frac{\Gamma \rhd M : \sigma}{\Gamma \rhd (\Lambda X.\, M) \colon \forall X.\, \sigma} \quad \text{(type abstraction)}$$

provided that X does not occur free in any of the types in Γ, and

$$\frac{\Gamma \rhd M : \forall X.\, \sigma}{\Gamma \rhd (M\tau) \colon \sigma[X := \tau]} \quad \text{(type application)}$$

where τ is any type (and no capture of variable takes place).

From the point of view where types are viewed as propositions and λ-terms are viewed as proofs, type abstraction is an introduction rule and type application is an elimination rule, both for the second-order quantifier $\forall$.

We also have a new reduction rule

$$(\Lambda X.\, M)\sigma \longrightarrow_{\beta\forall} M[X := \sigma]$$

that corresponds to a new form of redundancy in proofs having to do with a $\forall$-elimination immediately following a $\forall$-introduction. Here in the substitution $M[X := \tau]$, all free occurrences of X in M and the types in M are replaced by τ. For example,

$$\begin{aligned}(\Lambda X.\, \lambda f \colon (X \to X).\, \lambda x \colon X.\, \lambda g \colon \forall Y.\, (Y \to Y).\, gX\, fx)[X := \tau]& \\ = \lambda f \colon (\tau \to \tau).\, \lambda x \colon \tau.\, \lambda g \colon \forall Y.\, (Y \to Y).\, g\tau\, xf.&\end{aligned}$$

[3] Barendregt and others used Greek variables to denote type variables but we find this confusing.

For technical details, see Gallier [17].

This new typed λ-calculus is the *second-order polymorphic lambda calculus*. It was invented by Girard (1972) who named it *system F*; see Girard [22, 23], and it is denoted $\boldsymbol{\lambda 2}$ by Barendregt. From the point of view of logic, Girard's system is a proof system for *intuitionistic second-order propositional logic*. We define $\xrightarrow{+}_{\boldsymbol{\lambda 2}}$ and $\xrightarrow{*}_{\boldsymbol{\lambda 2}}$ as the relations

$$\xrightarrow{+}_{\boldsymbol{\lambda 2}} = (\longrightarrow_\beta \cup \longrightarrow_{\beta\forall})^+$$
$$\xrightarrow{*}_{\boldsymbol{\lambda 2}} = (\longrightarrow_\beta \cup \longrightarrow_{\beta\forall})^*.$$

A variant of system F was also introduced independently by John Reynolds (1974) but for very different reasons.

The intuition behind terms of type $\forall X.\,\sigma$ is that a term M of type $\forall X.\,\sigma$ is a sort of *generic function* such that for *any* type τ, the function $M\tau$ is a *specialized version* of type $\sigma[X := \tau]$ of M. For example, M could be the function that appends an element to a list, and for specific types such as the natural numbers Nat, strings String, trees Tree, *etc.*, the functions $M\mathsf{Nat}$, $M\mathsf{String}$, $M\mathsf{Tree}$, are the specialized versions of M to lists of elements having the specific data types $\mathsf{Nat}, \mathsf{String}, \mathsf{Tree}$.

For example, if σ is any type, we have the closed term

$$\mathbf{A}_\sigma = \lambda x\colon \sigma.\, \lambda f\colon (\sigma \to \sigma).\, fx,$$

of type $\sigma \to ((\sigma \to \sigma) \to \sigma)$, such that for every term F of type $\sigma \to \sigma$ and every term a of type σ,

$$\mathbf{A}_\sigma Fa \xrightarrow{+}_{\boldsymbol{\lambda 2}} Fa.$$

Since $\mathbf{A}_\sigma$ has the same behavior for *all* types σ, it is natural to define the generic function $\mathbf{A}$ given by

$$\mathbf{A} = \Lambda X.\, \lambda x\colon X.\, \lambda f\colon (X \to X).\, fx,$$

which has type $\forall X.\,(X \to ((X \to X) \to X))$, and then $\mathbf{A}\sigma$ has the same behavior as $\mathbf{A}_\sigma$. We will see shortly that $\mathbf{A}$ is the Church numeral $\mathbf{c}_1$ in $\boldsymbol{\lambda 2}$.

Remarkably, system F is *strongly normalizing*, which means that every λ-term typable in system F has a β-normal form. The proof of this theorem is hard and was one of Girard's accomplishments in his dissertation, Girard [23]. The Church–Rosser property also holds for system F. The proof technique used to prove that system F is strongly normalizing is thoroughly analyzed in Gallier [17].

We stated earlier that deciding whether a simple type σ is provable, that is, whether there is a closed λ-term M that type-checks in the simply-typed λ-calculus such that the judgement $\rhd M\colon \sigma$ is provable is a hard problem. Indeed Statman proved that this problem is P-space complete; see Statman [46] and Section 13.4.

It is natural so ask whether it is decidable whether given any second-order type σ, there is a closed λ-term M that type-checks in system F such that the judgement $\rhd M\colon \sigma$ is provable (if σ is viewed as a second-order logical formula, the problem is to decide whether σ

is provable). Surprisingly the answer is *no*; this problem (called *inhabitation*) is undecidable. This result was proven by Löb around 1976, see Barendregt [5].

This undecidability result is troubling and at first glance seems paradoxical. Indeed, viewed as a logical formula, a second-order type σ is a QBF (a quantified boolean formula), and if we assign the truth values $\mathbf{F}$ and $\mathbf{T}$ to the boolean variables in it, we can decide whether such a proposition is valid in exponential time and polynomial space (in fact, we will see that later QBF validity is P-space complete). This seems in contradiction with the fact that provability is undecidable.

But the proof system corresponding to system F is an *intuitionistic* proof system, so there are (non-quantifed) propositions that are valid in the truth-value semantics but not provable in intuitionistic propositional logic. The set of second-order propositions provable in intuitionistic second-order logic is a *proper* subset of the set of valid QBF (under the truth-value semantics), and it is *not computable*. So there is no paradox after all.

Going back to the issue of computability of numerical functions, a version of the *Church numerals* can be defined as

$$\mathbf{c}_n = \Lambda X.\, \lambda x\colon X.\, \lambda f\colon (X \to X).\, f^n(x). \tag{$*_{c1}$}$$

Observe that $\mathbf{c}_n$ has type Nat. Inspired by the definition of **Succ** given in Section 6.4, we can define the successor function on the natural numbers as

$$\mathbf{Succ} = \lambda n\colon \mathsf{Nat}.\, \Lambda X.\, \lambda x\colon X.\, \lambda f\colon (X \to X).\, f(nX\, xf).$$

Note how n, which is of type $\mathsf{Nat} = \forall X.\, (X \to ((X \to X) \to X))$, is applied to the type variable X in order to become a term nX of type $X \to ((X \to X) \to X)$, so that $nX\, xf$ has type X, thus $f(nX\, xf)$ also has type X.

For every type σ, every term F of type $\sigma \to \sigma$ and every term a of type σ, we have

$$\begin{aligned} \mathbf{c}_n\sigma\, aF &= \big(\Lambda X.\, \lambda x\colon X.\, \lambda f\colon (X \to X).\, f^n(x)\big)\sigma\, aF \\ &\xrightarrow{+}_{\boldsymbol{\lambda 2}} \big(\lambda x\colon \sigma.\, \lambda f\colon (\sigma \to \sigma).\, f^n(x)\big)aF \\ &\xrightarrow{+}_{\boldsymbol{\lambda 2}} F^n(a); \end{aligned}$$

that is,

$$\mathbf{c}_n\sigma\, aF \xrightarrow{+}_{\boldsymbol{\lambda 2}} F^n(a). \tag{$*_{c2}$}$$

So $\mathbf{c}_n\sigma$ iterates F n times starting with a. As a consequence,

$$\begin{aligned} \mathbf{Succ}\, \mathbf{c}_n &= \big(\lambda n\colon \mathsf{Nat}.\, \Lambda X.\, \lambda x\colon X.\, \lambda f\colon (X \to X).\, f(nX\, xf)\big)\mathbf{c}_n \\ &\xrightarrow{+}_{\boldsymbol{\lambda 2}} \Lambda X.\, \lambda x\colon X.\, \lambda f\colon (X \to X).\, f(\mathbf{c}_n X\, xf) \\ &\xrightarrow{+}_{\boldsymbol{\lambda 2}} \Lambda X.\, \lambda x\colon X.\, \lambda f\colon (X \to X).\, f(f^n(x)) \\ &= \Lambda X.\, \lambda x\colon X.\, \lambda f\colon (X \to X).\, f^{n+1}(x) = \mathbf{c}_{n+1}. \end{aligned}$$

We can also define addition of natural numbers as

$$\mathbf{Add} = \lambda m\colon \mathsf{Nat}.\, \lambda n\colon \mathsf{Nat}.\, \Lambda X.\, \lambda x\colon X.\, \lambda f\colon (X \to X).\, (mX\, f(nX\, xf))f.$$

Note how m and n, which are of type $\mathsf{Nat} = \forall X.\, (X \to ((X \to X) \to X))$, are applied to the type variable X in order to become terms mX and nX of type $X \to ((X \to X) \to X)$, so that $nX\, xf$ has type X, thus $f(nX\, xf)$ also has type X, and $mX\, f(nX\, xf)$ has type $(X \to X) \to X$, and finally $(mX\, f(nX\, xf))f$ has type X.

Many of the constructions that can be performed in the pure λ-calculus can be mimicked in system F, which explains its expressive power. For example, for any two second-order types σ and τ, we can define a pairing function $\langle -, - \rangle$ (to be very precise, $\langle -, - \rangle_{\sigma,\tau}$) given by

$$\langle -, - \rangle = \lambda u\colon \sigma.\, \lambda v\colon \tau.\, \Lambda X.\, \lambda f\colon \sigma \to (\tau \to X).\, fuv,$$

of type $\sigma \to \big(\tau \to \big(\forall X.\, ((\sigma \to (\tau \to X)) \to X)\big)\big)$. Given any term M of type σ and any term N of type τ, we have

$$\langle -, - \rangle_{\sigma,\tau} MN \xrightarrow{*}_{\boldsymbol{\lambda 2}} \Lambda X.\, \lambda f\colon \sigma \to (\tau \to X).\, fMN.$$

Thus we define $\langle M, N \rangle$ as

$$\langle M, N \rangle = \Lambda X.\, \lambda f\colon \sigma \to (\tau \to X).\, fMN,$$

and the type

$$\forall X.\, ((\sigma \to (\tau \to X)) \to X)$$

of $\langle M, N \rangle$ is denoted by $\sigma \times \tau$. As a logical formula it is equivalent to $\sigma \wedge \tau$, which means that if we view σ and τ as (second-order) propositions, then

$$\sigma \wedge \tau \equiv \forall X.\, ((\sigma \to (\tau \to X)) \to X)$$

is provable intuitionistically. This is a special case of the result that we mentioned earlier: the connectives $\wedge, \vee, \neg$ and $\exists$ are definable in terms of $\to$ (as a logical connective, $\Rightarrow$) and $\forall$.

Proposition 6.13. *The connectives $\wedge, \vee, \neg, \perp$ and $\exists$ are definable in terms of $\to$ and $\forall$, which means that the following equivalences are provable intuitionistically, where X is not free in σ or τ:*

$$\begin{aligned}
\sigma \wedge \tau &\equiv \forall X.\, \big((\sigma \to (\tau \to X)) \to X\big) \\
\sigma \vee \tau &\equiv \forall X.\, \big((\sigma \to X) \to ((\tau \to X) \to X)\big) \\
\perp &\equiv \forall X.\, X \\
\neg\sigma &\equiv \sigma \to \forall X.\, X \\
\exists Y.\, \sigma &\equiv \forall X.\, \big((\forall Y.\, (\sigma \to X)) \to X\big).
\end{aligned}$$

We leave the proof as an exercise, or see Troelstra and Schwichtenberg [50] (Chapter 11).

Remark: The rule of type application implies that $\perp \rightarrow \sigma$ is intuitionistically provable for *all* propositions (types) σ. So in second-order logic there is no difference between minimal and intuitionistic logic.

We also have two projections π_1 and π_2 (to be very precise $\pi_1^{\sigma\times\tau}$ and $\pi_2^{\sigma\times\tau}$) given by

$$
\begin{aligned}
\pi_1 &= \lambda g\colon \sigma \times \tau.\, g\sigma(\lambda x\colon \sigma.\, \lambda y\colon \tau.\, x) \\
\pi_2 &= \lambda g\colon \sigma \times \tau.\, g\tau(\lambda x\colon \sigma.\, \lambda y\colon \tau.\, y).
\end{aligned}
$$

It is easy to check that π_1 has type $(\sigma \times \tau) \rightarrow \sigma$ and that π_2 has type $(\sigma \times \tau) \rightarrow \tau$. The reader should check that for any M of type σ and any N of type τ we have

$$\pi_1\langle M, N\rangle \xrightarrow{+}_{\boldsymbol{\lambda 2}} M \quad \text{and} \quad \pi_2\langle M, N\rangle \xrightarrow{+}_{\boldsymbol{\lambda 2}} N.$$

For example, we have

$$
\begin{aligned}
\pi_1\langle M, N\rangle &= \big(\lambda g\colon \sigma \times \tau.\, g\sigma(\lambda x\colon \sigma.\, \lambda y\colon \tau.\, x)\big)\big(\Lambda X.\, \lambda f\colon \sigma \rightarrow (\tau \rightarrow X).\, fMN\big) \\
&\xrightarrow{+}_{\boldsymbol{\lambda 2}} \big(\Lambda X.\, \lambda f\colon \sigma \rightarrow (\tau \rightarrow X).\, fMN\big)\sigma(\lambda x\colon \sigma.\, \lambda y\colon \tau.\, x) \\
&\xrightarrow{+}_{\boldsymbol{\lambda 2}} \big(\lambda f\colon \sigma \rightarrow (\tau \rightarrow \sigma).\, fMN\big)(\lambda x\colon \sigma.\, \lambda y\colon \tau.\, x) \\
&\xrightarrow{+}_{\boldsymbol{\lambda 2}} (\lambda x\colon \sigma.\, \lambda y\colon \tau.\, x)MN \\
&\xrightarrow{+}_{\boldsymbol{\lambda 2}} (\lambda y\colon \tau.\, M)N \\
&\xrightarrow{+}_{\boldsymbol{\lambda 2}} M.
\end{aligned}
$$

The booleans can be defined as

$$
\begin{aligned}
\mathbf{T} &= \Lambda X.\, \lambda x\colon X.\, \lambda y\colon X.\, x \\
\mathbf{F} &= \Lambda X.\, \lambda x\colon X.\, \lambda y\colon X.\, y,
\end{aligned}
$$

both of type $\mathsf{Bool} = \forall X.\, (X \rightarrow (X \rightarrow X))$. We also define $\mathsf{if\ then\ else}$ as

$$\mathsf{if\ then\ else} = \Lambda X.\, \lambda z\colon \mathsf{Bool}.\, zX$$

of type $\forall X.\, \mathsf{Bool} \rightarrow (X \rightarrow (X \rightarrow X))$. It is easy that for any type σ and any two terms M and N of type σ we have

$$
\begin{aligned}
(\mathsf{if}\ \mathbf{T}\ \mathsf{then}\ M\ \mathsf{else}\ N)\sigma &\xrightarrow{+}_{\boldsymbol{\lambda 2}} M \\
(\mathsf{if}\ \mathbf{F}\ \mathsf{then}\ M\ \mathsf{else}\ N)\sigma &\xrightarrow{+}_{\boldsymbol{\lambda 2}} N,
\end{aligned}
$$

where we write $(\mathsf{if}\ \mathbf{T}\ \mathsf{then}\ M\ \mathsf{else}\ N)\sigma$ instead of $(\mathsf{if\ then\ else})\ \sigma\, \mathbf{T} M N$ (and similarly for the other term). For example, we have

$$\begin{aligned}(\mathsf{if}\ \mathbf{T}\ \mathsf{then}\ M\ \mathsf{else}\ N)\sigma &= \big(\Lambda X.\, \lambda z\colon \mathsf{Bool}.\, zX\big)\sigma\, \mathbf{T} M N\\ &\xrightarrow{+}_{\boldsymbol{\lambda 2}} \big(\lambda z\colon \mathsf{Bool}.\, z\sigma\big)\mathbf{T} M N\\ &\xrightarrow{+}_{\boldsymbol{\lambda 2}} \big(\mathbf{T}\sigma\big) M N\\ &= \big((\Lambda X.\, \lambda x\colon X.\, \lambda y\colon X.\, x)\sigma\big) M N\\ &\xrightarrow{+}_{\boldsymbol{\lambda 2}} \big(\lambda x\colon \sigma.\, \lambda y\colon \sigma.\, x\big) M N\\ &\xrightarrow{+}_{\boldsymbol{\lambda 2}} M.\end{aligned}$$

Lists, trees, and other inductively data stuctures are also representable in system F; see Girard–Lafond–Taylor [21].

We can also define an iterator **Iter** given by

$$\mathbf{Iter} = \Lambda X.\, \lambda u\colon X.\, \lambda f\colon (X \to X).\, \lambda z\colon \mathsf{Nat}.\, zX\, uf$$

of type $\forall X.\, (X \to ((X \to X) \to (\mathsf{Nat} \to X)))$. The idea is that given f of type $\sigma \to \sigma$ and u of type σ, the term $\mathbf{Iter}\, \sigma\, u f \mathbf{c}_n$ iterates f n times over the input u. It is easy to show that for any term t of type Nat we have

$$\begin{aligned}\mathbf{Iter}\, \sigma\, u f \mathbf{c}_0 &\xrightarrow{+}_{\boldsymbol{\lambda 2}} u\\ \mathbf{Iter}\, \sigma\, u f (\mathbf{Succ}_{\mathbf{c}}\, t) &\xrightarrow{+}_{\boldsymbol{\lambda 2}} f(\mathbf{Iter}\, \sigma\, u f t),\end{aligned}$$

and that

$$\mathbf{Iter}\, \sigma\, u f \mathbf{c}_n \xrightarrow{+}_{\boldsymbol{\lambda 2}} f^n(u).$$

Then mimicking what we did in the pure λ-calculus, we can show that the primitive recursive functions are λ-definable in system F. Actually, higher-order primitive recursion is definable. So, for example, Ackermann's function is definable.

Remarkably, the class of numerical functions definable in system F is a class of (total) computable functions much bigger than the class of primitive recursive functions. This class of functions was characterized by Girard as the functions that are provably-recursive in a formalization of arithmetic known as *intuitionistic second-order arithmetic*; see Girard [23], Troelstra and Schwichtenberg [50] and Girard–Lafond–Taylor [21]. It can also be shown (using a diagonal argument) that there are (total) computable functions not definable in system F.

From a theoretical point of view, every (total) function that we will ever want to compute is definable in system F. However, from a practical point of view, programming in system F is very tedious and usually leads to very inefficient programs. Nevertheless polymorphism is an interesting paradigm which had made its way in certain programming languages.

Type systems even more powerful than system F have been designed, the ultimate system being the *calculus of constructions* due to Huet and Coquand, but these topics are beyond the scope of these notes.

One last comment has to do with the use of the simply-typed λ-calculus as a the core of a programming language. In the early 1970's Dana Scott defined a system named LCF based on the the simply-typed λ-calculus and obtained by adding the natural numbers and the booleans as data types, product types, and a fixed-point operator. Robin Milner then extended LCF, and as a by-product, defined a programming language known as ML, which is the ancestor of most functional programming languages. A masterful and thorough exposition of type theory and its use in programming language design is given in Pierce [39].

We now revisit the problem of defining the partial computable functions.

6.8 Head Normal-Forms and the Partial Computable Functions

One defect of the proof of Theorem 6.11 in the case where a computable function is partial is the use of the Kleene normal form. The difficulty has to do with composition. Given a partial computable function g λ-defined by a closed term G and a partial computable function h λ-defined by a closed term H (for simplicity we assume that both g and h have a single argument), it would be nice if the composition $h \circ g$ was represented by $\lambda x.\, H(Gx)$. This is true if both g and h are total, but false if either g or h is partial as shown by the following example from Barendregt [4] (Chapter 2, §2). If g is the function undefined everywhere and h is the constant function 0, then g is λ-defined by $G = \mathbf{K\Omega}$ and h is λ-defined by $H = \mathbf{K}\,\mathbf{c}_0$, with $\mathbf{\Omega} = (\lambda x.\,(xx))(\lambda x.\,(xx))$. We have

$$\lambda x.\, H(Gx) = \lambda x.\, \mathbf{K}\,\mathbf{c}_0(\mathbf{K\Omega}x) \xrightarrow{+}_{\beta} \lambda x.\, \mathbf{K}\mathbf{c}_0\mathbf{\Omega} \xrightarrow{+}_{\beta} \lambda x.\, \mathbf{c}_0,$$

but $h \circ g = g$ is the function undefined everywhere, and $\lambda x.\, \mathbf{c}_0$ represents the total function h, so $\lambda x.\, H(Gx)$ *does not* λ-define $h \circ g$.

It turns out that the λ-definability of the partial computable functions can be obtained in a more elegant fashion without having recourse to the Kleene normal form by capturing the fact that a function is undefined for some input is a more subtle way. The key notion is the notion of *head normal form*, which is more general than the notion of β-normal form. As a consequence, there a *fewer* λ-terms having *no* head normal form than λ-terms having *no* β-normal form, and we capture a stronger form of divergence.

Recall that a λ-term is either a variable x, or an application (MN), or a λ-abstraction $(\lambda x.\, M)$. We can sharpen this characterization as follows.

Proposition 6.14. *The following properties hold:*

(1) Every application term M is of the form

$$M = (N_1 N_2 \cdots N_{n-1})N_n, \quad n \geq 2,$$

where N_1 is not an application term.

(2) *Every abstraction term M is of the form*

$$M = \lambda x_1 \cdots x_n.\, N, \quad n \geq 1,$$

where N is not an abstraction term.

(3) *Every λ-term M is of one of the following two forms:*

$$M = \lambda x_1 \cdots x_n.\, x M_1 \cdots M_m, \quad m, n \geq 0 \tag{a}$$

$$M = \lambda x_1 \cdots x_n.\, (\lambda x.\, M_0) M_1 \cdots M_m, \quad m \geq 1, n \geq 0, \tag{b}$$

where x is a variable.

Proof. (1) Suppose that M is an application $M = M_1 M_2$. We proceed by induction on the depth of M_1. For the base case M_1 must be variables and we are done. For the induction step, if M_1 is a λ-abstraction, we are done. If M_1 is an application, then by the induction hypothesis it is of the form

$$M_1 = (N_1 N_2 \cdots N_{n-1}) N_n, \quad n \geq 2,$$

where N_1 is not an application term, and then

$$M = M_1 M_2 = ((N_1 N_2 \cdots N_{n-1}) N_n) M_2 \quad n \geq 2,$$

where N_1 is not an application term.

The proof of (2) is similar.

(3) We proceed by induction on the depth of M. If M is a variable, then we are in Case (a) with $m = n = 0$.

If M is an application, then by (1) it is of the form $M = N_1 N_2 \cdots N_p$ with N_1 not an application term. This means that either N_1 is a variable, in which case we are in Case (a) with $n = 0$, or N_1 is an abstraction, in which case we are in Case (b) also with $n = 0$.

If M is an abstraction $\lambda x.\, N$, then by the induction hypothesis N is of the form (a) or (b), and by adding one more binder λx in front of these expressions we preserve the shape of (a) and (b) by increasing n by 1. $\square$

The terms, $\mathbf{I}, \mathbf{K}, \mathbf{K}_*, \mathbf{S}$, the Church numerals $\mathbf{c}_n$, $\mathsf{if\ then\ else}$, $\langle M, N\rangle$, π_1, π_2, **Iter**, $\mathbf{Succ_c}$, **Add** and **Mult** as in Proposition 6.5, are λ-terms of type (a). However, $\mathbf{Pred_K}$, $\mathbf{\Omega} = (\lambda x.\, (xx))(\lambda x.\, (xx))$, $\mathbf{Y}$ (the Curry $\mathbf{Y}$-combinator), $\mathbf{\Theta}$ (the Turing $\mathbf{\Theta}$-combinator) are of type (b).

Proposition 6.14 motivates the following definition.

Definition 6.17. A λ-term M is a *head normal form* (for short *hnf*) if it is of the form (a), namely

$$M = \lambda x_1 \cdots x_n.\, x M_1 \cdots M_m, \quad m, n \geq 0,$$

where x is a variable called the *head variable*.

A λ-term M *has a head normal form* if there is some head normal form N such that $M \xrightarrow{*}_\beta N$.

In a term M of the form (b),

$$M = \lambda x_1 \cdots x_n.\, (\lambda x.\, M_0)M_1 \cdots M_m, \quad m \geq 1, n \geq 0,$$

the subterm $(\lambda x.\, M_0)M_1$ is called the *head redex* of M.

In addition to the terms of type (a) that we listed after Proposition 6.14, the term $\lambda x.\, x\mathbf{\Omega}$ is a head normal form. It is the head normal form of the term $\lambda x.\, (\mathbf{I}x)\mathbf{\Omega}$, which has *no* β-normal form.

Not every term has a head normal form. For example, the term

$$\mathbf{\Omega} = (\lambda x.\, (xx))(\lambda x.\, (xx))$$

has no head normal form. Every β-normal form must be a head normal form, but the converse is false as we saw with

$$M = \lambda x.\, x\mathbf{\Omega},$$

which is a head normal form but has no β-normal form.

Note that a head redex of a term is a leftmost redex, but not conversely, as shown by the term $\lambda x.\, x((\lambda y.\, y)x)$.

A term may have more than one head normal form but here is a way of obtaining a head normal form (if there is one) in a systematic fashion.

Definition 6.18. The relation $\longrightarrow_h$, called *one-step head reduction*, is defined as follows: For any two terms M and N, if M contains a head redex $(\lambda x.\, M_0)M_1$, which means that M is of the form

$$M = \lambda x_1 \cdots x_n.\, (\lambda x.\, M_0)M_1 \cdots M_m, \quad m \geq 1, n \geq 0,$$

then $M \longrightarrow_h N$ with

$$N = \lambda x_1 \cdots x_n.\, (M_0[x := M_1])M_2 \cdots M_m.$$

We denote by $\xrightarrow{+}_h$ the transitive closure of $\longrightarrow_h$ and by $\xrightarrow{*}_h$ the reflexive and transitive closure of $\longrightarrow_h$.

Given a term M containing a head redex, the *head reduction sequence of* M is the uniquely determined sequence of one-step head reductions

$$M = M_0 \longrightarrow_h M_1 \longrightarrow_h \cdots \longrightarrow_h M_n \longrightarrow_h \cdots.$$

If the head reduction sequence reaches a term M_n which is a head normal form we say that the sequence *terminates*, and otherwise we say that M has an *infinite* head reduction.

The following result is shown in Barendregt [4] (Chapter 8, §3).

Theorem 6.15. *(Wadsworth) A λ-term M has a head normal form if and only if the head reduction sequence terminates.*

In some intuitive sense, a λ-term M that does not have any head normal form has a strong divergence behavior with respect to β-reduction.

Remark: There is a notion more general than the notion of head normal form which comes up in functional languages (for example, Haskell). A λ-term M is a *weak head normal form* if it of one of the two forms

$$\lambda x.\, N \quad \text{or} \quad yN_1 \cdots N_m$$

where y is a variable These are exactly the terms that do not have a redex of the form $(\lambda x.\, M_0)M_1N_1 \cdots N_m$. Every head normal form is a weak head normal form, but there are many more weak head normal forms than there are head normal forms since a term of the form $\lambda x.\, N$ where N is *arbitrary* is a weak head normal form, but not a head normal form unless N is of the form $\lambda x_1 \cdots x_n.\, xM_1 \cdots M_m$, with $m, n \geq 0$.

Reducing to a weak head normal form is a lazy evaluation strategy.

There is also another useful notion which turns out to be equivalent to having a head normal form.

Definition 6.19. A closed λ-term M is *solvable* if there are closed terms $N_1, \ldots, N_n$ such that

$$MN_1 \cdots N_n \xrightarrow{*}_\beta \mathbf{I}.$$

A λ-term M with free variables $x_1, \ldots, x_m$ is *solvable* if the closed term $\lambda x_1 \cdots x_m.\, M$ is solvable. A term is *unsolvable* if it is not solvable.

The following result is shown in Barendregt [4] (Chapter 8, §3).

Theorem 6.16. *(Wadsworth) A λ-term M has a head normal form if and only if is it solvable.*

Actually, the proof that having a head normal form implies solvable is not hard.

We are now ready to revise the notion of λ-definability of numerical functions.

Definition 6.20. A function (partial or total) $f \colon \mathbb{N}^n \to \mathbb{N}$ is *strongly λ-definable* if for all $m_1, \ldots, m_n \in \mathbb{N}$, there is a combinator (a closed λ-term) F with the following properties:

(1) If the value $f(m_1, \ldots, m_n)$ is defined, then $F\mathbf{c}_{m_1} \cdots \mathbf{c}_{m_n}$ reduces to the β-normal form $\mathbf{c}_{f(m_1,\ldots,m_n)}$.

(2) If $f(m_1, \ldots, m_n)$ is undefined, then $F\mathbf{c}_{m_1} \cdots \mathbf{c}_{m_n}$ has no head normal form, or equivalently, is unsolvable.

Observe that in Case (2), when the value $f(m_1, \ldots, m_n)$ is undefined, the divergence behavior of $F\mathbf{c}_{m_1} \cdots \mathbf{c}_{m_n}$ is stronger than in Definition 6.16. Not only $F\mathbf{c}_{m_1} \cdots \mathbf{c}_{m_n}$ has no β-normal form, but actually it has *no head normal form*.

The following result is proven in Barendregt [4] (Chapter 8, §4). The proof does not use the Kleene normal form. Instead, it makes clever use of the term **KII**. Another proof is given in Krivine [33] (Chapter II).

Theorem 6.17. *Every partial or total computable function is strongly λ-definable. Conversely, every strongly λ-definable function is partial computable.*

Making sure that a composition $g \circ (h_1, \ldots, h_m)$ is defined for some input $x_1, \ldots, x_n$ iff all the $h_i(x_1, \ldots, x_n)$ and $g(h_1(x_1, \ldots, x_n), \ldots, h_m(x_1, \ldots, x_n))$ are defined is tricky. The term **KII** comes to the rescue! If g is strongly λ-definable by G and the h_i are strongly λ-definable by H_i, then it can be shown that the combinator F given by

$$F = \lambda x_1 \cdots x_n.\, (H_1 x_1 \cdots x_n \mathbf{KII}) \cdots (H_m x_1 \cdots x_n \mathbf{KII})(G(H_1 x_1 \cdots x_n) \cdots (G(H_m x_1 \cdots x_n))$$

strongly λ-defines F; see Barendregt [4] (Chapter 8, Lemma 8.4.6).

Chapter 7

Recursion Theory; More Advanced Topics

This chapter is devoted to three advanced topics of recursion theory:

(1) The recursion theorem.

(2) The extended Rice theorem.

(3) Creative and productive sets and their use in proving a strong version of Gödel's first incompleteness theorem.

The recursion theorem is a deep result and an important technical tool in recursion theory.

The extended Rice theorem gives a characterization of the sets of partial computable functions that are listable in terms of extensions of partial computable functions with finite domains.

Productive and creative sets arise when dealing with truth and provability in arithmetic. The "royal road" to Gödel's first incompleteness theorem is to first prove that for any proof system for arithmetic that only proves true statements (and is rich enough), the set of true sentences of arithmetic is productive. Productive sets are *not* listable in a strong sense, so we deduce that it is impossible to axiomatize the set of true sentences of arithmetic in a computable manner. The set of provable sentences of arithmetic is creative, which implies that it is impossible to decide whether a sentence of arithmetic is provable. This also implies that there are true sentences F such that neither F nor $\neg F$ are provable.

7.1 The Recursion Theorem

The recursion theorem, due to Kleene, is a fundamental result in recursion theory. Let f be a total computable function. Then it turns out that there is some n such that

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

To understand why such a mysterious result is interesting, consider the recursive definition of the factorial function $fact(n) = n!$ given by

$$\begin{aligned} fact(0) &= 1 \\ fact(n+1) &= (n+1)fact(n). \end{aligned}$$

The trick is to define the partial computable computable function g (defined on $\mathbb{N}^2$) given by

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

for all $m, n \in \mathbb{N}$. By the s-m-n Theorem, there is a computable function f such that

$$g(m,n) = \varphi_{f(m)}(n) \quad \text{for all } m, n \in \mathbb{N}.$$

Then the equations above become

$$\begin{aligned} \varphi_{f(m)}(0) &= 1 \\ \varphi_{f(m)}(n+1) &= (n+1)\varphi_m(n). \end{aligned}$$

Since f is (total) recursive, there is some m_0 such that $\varphi_{m_0} = \varphi_{f(m_0)}$, and for m_0 we get

$$\begin{aligned} \varphi_{m_0}(0) &= 1 \\ \varphi_{m_0}(n+1) &= (n+1)\varphi_{m_0}(n), \end{aligned}$$

so the partial recursive function φ_{m_0} *satisfies the recursive definition of factorial*, which means that it is a *fixed point* of the recursive equations defining factorial. Since factorial is a total function, $\varphi_{m_0} = fact$, that is, factorial is a total computable function.

More generally, if a function h (over $\mathbb{N}^k$) is defined in terms of recursive equations of the form

$$h(z_1, \ldots, z_k) = t(h(y_1, \ldots, y_k))$$

where $y_1, \ldots, y_k, z_1, \ldots, z_k$ are expressions in some variables $x_1, \ldots, x_k$ ranging over $\mathbb{N}$ and where t is an expression containing recursive occurrences of h, if we can show that the equations

$$g(m, z_1, \ldots, z_k) = t(\varphi_m(y_1, \ldots, y_k))$$

define a partial computable function g, then we can use the above trick to put them in the form

$$\varphi_{f(m)}(z_1, \ldots, z_k) = t(\varphi_m(y_1, \ldots, y_k)).$$

for some computable function f. Such a formalism is decribed in detail in Chapter XI of Kleene I.M [32]. By the recursion theorem, there is some m_0 such that $\varphi_{m_0} = \varphi_{f(m_0)}$, so φ_{m_0} satisfies the recursive equations

$$\varphi_{m_0}(z_1, \ldots, z_k) = t(\varphi_{m_0}(y_1, \ldots, y_k)),$$

and φ_{m_0} is a fixed point of these recursive equations. If we can show that φ_{m_0} is total, then we found *the* fixed point of this set of recursive equations and $h = \varphi_{m_0}$ is a total computable function. If φ_{m_0} is a partial function, it is still a fixed point. However in general there is more than one fixed point and we don't which one φ_{m_0} is (it could be the partial function undefined everywhere).

Theorem 7.1. *(Recursion Theorem, Version 1) Let $\varphi_0, \varphi_1, \ldots$ be any acceptable indexing of the partial computable functions. For every total computable function f, there is some n such that*

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

Proof. Consider the function θ defined such that

$$\theta(x, y) = \varphi_{univ}(\varphi_{univ}(x, x), y) \quad \text{for all } x, y \in \mathbb{N}.$$

The function θ is partial computable, and there is some index j such that $\varphi_j = \theta$. By the s-m-n Theorem, there is a computable function g such that

$$\varphi_{g(x)}(y) = \theta(x, y).$$

Consider the function $f \circ g$. Since it is computable, there is some index m such that $\varphi_m = f \circ g$. Let

$$n = g(m).$$

Since φ_m is total, $\varphi_m(m)$ is defined, and we have

$$\begin{aligned}\varphi_n(y) = \varphi_{g(m)}(y) = \theta(m, y) &= \varphi_{univ}(\varphi_{univ}(m, m), y) = \varphi_{\varphi_{univ}(m,m)}(y) \\ &= \varphi_{\varphi_m(m)}(y) = \varphi_{f \circ g(m)}(y) = \varphi_{f(g(m))}(y) = \varphi_{f(n)}(y),\end{aligned}$$

for all $y \in \mathbb{N}$. Therefore, $\varphi_n = \varphi_{f(n)}$, as desired. $\square$

The recursion theorem can be strengthened as follows.

Theorem 7.2. *(Recursion Theorem, Version 2) Let $\varphi_0, \varphi_1, \ldots$ be any acceptable indexing of the partial computable functions. There is a total computable function h such that for all $x \in \mathbb{N}$, if φ_x is total, then*

$$\varphi_{\varphi_x(h(x))} = \varphi_{h(x)}.$$

Proof. The computable function g obtained in the proof of Theorem 7.1 satisfies the condition

$$\varphi_{g(x)} = \varphi_{\varphi_x(x)},$$

and it has some index i such that $\varphi_i = g$. Recall that c is a computable composition function such that

$$\varphi_{c(x,y)} = \varphi_x \circ \varphi_y.$$

It is easily verified that the function h defined such that

$$h(x) = g(c(x, i)) \quad \text{for all } x \in \mathbb{N}$$

does the job. $\square$

A third version of the recursion Theorem is given below.

Theorem 7.3. *(Recursion Theorem, Version 3) For all $n \geq 1$, there is a total computable function h of $n+1$ arguments, such that for all $x \in \mathbb{N}$, if φ_x is a total computable function of $n+1$ arguments, then*

$$\varphi_{\varphi_x(h(x,x_1,\ldots,x_n),x_1,\ldots,x_n)} = \varphi_{h(x,x_1,\ldots,x_n)},$$

for all $x_1, \ldots, x_n \in \mathbb{N}$.

Proof. Let θ be the function defined such that

$$\theta(x, x_1, \ldots, x_n, y) = \varphi_{\varphi_x(x,x_1,\ldots,x_n)}(y) = \varphi_{univ}(\varphi_{univ}(x, x, x_1, \ldots, x_n), y)$$

for all $x, x_1, \ldots, x_n, y \in \mathbb{N}$. By the s-m-n Theorem, there is a computable function g such that

$$\varphi_{g(x,x_1,\ldots,x_n)} = \varphi_{\varphi_x(x,x_1,\ldots,x_n)}.$$

It is easily shown that there is a computable function c such that

$$\varphi_{c(i,j)}(x, x_1, \ldots, x_n) = \varphi_i(\varphi_j(x, x_1, \ldots, x_n), x_1, \ldots, x_n)$$

for any two partial computable functions φ_i and φ_j (viewed as functions of $n+1$ arguments) and all $x, x_1, \ldots, x_n \in \mathbb{N}$. Let $\varphi_i = g$, and define h such that

$$h(x, x_1, \ldots, x_n) = g(c(x, i), x_1, \ldots, x_n),$$

for all $x, x_1, \ldots, x_n \in \mathbb{N}$. We have

$$\varphi_{h(x,x_1,\ldots,x_n)} = \varphi_{g(c(x,i),x_1,\ldots,x_n)} = \varphi_{\varphi_{c(x,i)}(c(x,i),x_1,\ldots,x_n)},$$

and using the fact that $\varphi_i = g$,

$$\begin{aligned}
\varphi_{\varphi_{c(x,i)}(c(x,i),x_1,\ldots,x_n)} &= \varphi_{\varphi_x(\varphi_i(c(x,i),x_1,\ldots,x_n),x_1,\ldots,x_n)}, \\
&= \varphi_{\varphi_x(g(c(x,i),x_1,\ldots,x_n),x_1,\ldots,x_n)}, \\
&= \varphi_{\varphi_x(h(x,x_1,\ldots,x_n),x_1,\ldots,x_n)}.
\end{aligned}$$ □

As a first application of the recursion theorem, we can show that there is an index n such that φ_n is the constant function with output n. Loosely speaking, φ_n prints its own name. Let f be the computable function such that

$$f(x, y) = x$$

for all $x, y \in \mathbb{N}$. By the s-m-n Theorem, there is a computable function g such that

$$\varphi_{g(x)}(y) = f(x, y) = x$$

for all $x, y \in \mathbb{N}$. By the Theorem 7.1, there is some n such that

$$\varphi_{g(n)} = \varphi_n,$$

the constant function with value n.

As a second application, we get a very short proof of Rice's theorem. Let C be such that $P_C \neq \emptyset$ and $P_C \neq \mathbb{N}$, and let $j \in P_C$ and $k \in \mathbb{N} - P_C$. Define the function f as follows:

$$f(x) = \begin{cases} j & \text{if } x \notin P_C, \\ k & \text{if } x \in P_C, \end{cases}$$

If P_C is computable, then f is computable. By the recursion theorem (Theorem 7.1), there is some n such that

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

But then we have

$$n \in P_C \quad \text{iff} \quad f(n) \notin P_C$$

by definition of f, and thus,

$$\varphi_{f(n)} \neq \varphi_n,$$

a contradiction. Hence, P_C is not computable.

As a third application, we prove the following proposition.

Proposition 7.4. *Let C be a set of partial computable functions and let*

$$A = \{x \in \mathbb{N} \mid \varphi_x \in C\}.$$

The set A is not reducible to its complement $\overline{A}$.

Proof. Assume that $A \leq \overline{A}$. Then there is a computable function f such that

$$x \in A \quad \text{iff} \quad f(x) \in \overline{A}$$

for all $x \in \mathbb{N}$. By the recursion theorem, there is some n such that

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

But then,

$$\varphi_n \in C \quad \text{iff} \quad n \in A \quad \text{iff} \quad f(n) \in \overline{A} \quad \text{iff} \quad \varphi_{f(n)} \in \overline{C},$$

contradicting the fact that

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

∎

The recursion theorem can also be used to show that functions defined by recursive definitions other than primitive recursion are partial computable, as we discussed a the beginning of this section. This is the case for the function known as *Ackermann's function* discussed in Section 3.9 and defined recursively as follows:

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

It can be shown that this function is not primitive recursive. Intuitively, it outgrows all primitive recursive functions. However, f is computable, but this is not so obvious. We can use the recursion theorem to prove that f is computable. Using the technique described at the beginning of this section consider the following definition by cases:

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

Clearly, g is partial computable. By the s-m-n Theorem, there is a computable function h such that

$$\varphi_{h(n)}(x, y) = g(n, x, y).$$

The equations defining g yield

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

By the recursion theorem, there is an m such that

$$\varphi_{h(m)} = \varphi_m.$$

Therefore, the partial computable function $\varphi_m(x, y)$ satisfies the equations

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

defining Ackermann's function. We showed in Section 3.9 that $\varphi_m(x, y)$ is a total function, and thus, $f = \varphi_m$ and Ackermann's function is a total computable function.

Hence, the recursion theorem justifies the use of certain recursive definitions. However, note that there are some recursive definitions that are only satisfied by the completely undefined function.

In the next section, we prove the extended Rice theorem.

7.2 Extended Rice Theorem

The extended Rice theorem characterizes the sets of partial computable functions C such that P_C is listable (c.e., r.e.). First, we need to discuss a way of indexing the partial computable functions that have a finite domain. Using the uniform projection function Π (see Definition 4.3), we define the primitive recursive function F such that

$$F(x,y) = \Pi(y+1, \Pi_1(x)+1, \Pi_2(x)).$$

We also define the sequence of partial functions $P_0, P_1, \ldots$ as follows:

$$P_x(y) = \begin{cases} F(x,y) - 1 & \text{if } 0 < F(x,y) \text{ and } y < \Pi_1(x)+1, \\ \text{undefined} & \text{otherwise.} \end{cases}$$

Proposition 7.5. *Every P_x is a partial computable function with finite domain, and every partial computable function with finite domain is equal to some P_x.*

The proof is left as an exercise. The easy part of the extended Rice theorem is the following lemma. Recall that given any two partial functions $f\colon A \to B$ and $g\colon A \to B$, we say that g *extends* f iff $f \subseteq g$, which means that $g(x)$ is defined whenever $f(x)$ is defined, and if so, $g(x) = f(x)$.

Proposition 7.6. *Let C be a set of partial computable functions. If there is a listable (c.e., r.e.) set A such that $\varphi_x \in C$ iff there is some $y \in A$ such that φ_x extends P_y, then $P_C = \{x \mid \varphi_x \in C\}$ is listable (c.e., r.e.).*

Proof. Proposition 7.6 can be restated as

$$P_C = \{x \mid \exists y \in A,\ P_y \subseteq \varphi_x\}$$

is listable. If A is empty, so is P_C, and P_C is listable. Otherwise, let f be a computable function such that

$$A = range(f).$$

Let ψ be the following partial computable function:

$$\psi(z) = \begin{cases} \Pi_1(z) & \text{if } P_{f(\Pi_2(z))} \subseteq \varphi_{\Pi_1(z)}, \\ \text{undefined} & \text{otherwise.} \end{cases}$$

It is clear that

$$P_C = range(\psi).$$

To see that ψ is partial computable, write $\psi(z)$ as follows:

$$\psi(z) = \begin{cases} \Pi_1(z) & \text{if } \forall w \leq \Pi_1(f(\Pi_2(z))) \\ & [F(f(\Pi_2(z)), w) > 0 \Rightarrow \varphi_{\Pi_1(z)}(w) = F(f(\Pi_2(z)), w) - 1], \\ \text{undefined} & \text{otherwise.} \end{cases}$$

This completes the proof. ∎

To establish the converse of Proposition 7.6, we need two propositions.

Proposition 7.7. *If P_C is listable (c.e., r.e.) and $\varphi \in C$, then there is some $P_y \subseteq \varphi$ such that $P_y \in C$.*

Proof. Assume that P_C is listable and that $\varphi \in C$. By an s-m-n construction, there is a computable function g such that

$$\varphi_{g(x)}(y) = \begin{cases} \varphi(y) & \text{if } \forall z \leq y[\neg T(x, x, z)], \\ \text{undefined} & \text{if } \exists z \leq y[T(x, x, z)], \end{cases}$$

for all $x, y \in \mathbb{N}$. Observe that if $x \in K$, then $\varphi_{g(x)}$ is a finite subfunction of φ, and if $x \in \overline{K}$, then $\varphi_{g(x)} = \varphi$. Assume that no finite subfunction of φ is in C. Then

$$x \in \overline{K} \quad \text{iff} \quad g(x) \in P_C$$

for all $x \in \mathbb{N}$, that is, $\overline{K} \leq P_C$. Since P_C is listable, $\overline{K}$ would also be listable, a contradiction. $\square$

As a corollary of Proposition 7.7, we note that TOTAL is not listable.

Proposition 7.8. *If P_C is listable (c.e., r.e.), $\varphi \in C$, and $\varphi \subseteq \psi$, where ψ is a partial computable function, then $\psi \in C$.*

Proof. Assume that P_C is listable. We claim that there is a computable function h such that

$$\varphi_{h(x)}(y) = \begin{cases} \psi(y) & \text{if } x \in K, \\ \varphi(y) & \text{if } x \in \overline{K}, \end{cases}$$

for all $x, y \in \mathbb{N}$. Assume that $\psi \notin C$. Then

$$x \in \overline{K} \quad \text{iff} \quad h(x) \in P_C$$

for all $x \in \mathbb{N}$, that is, $\overline{K} \leq P_C$, a contradiction, since P_C is listable. Therefore, $\psi \in C$. To find the function h we proceed as follows: Let $\varphi = \varphi_j$ and define Θ such that

$$\Theta(x, y, z) = \begin{cases} \varphi(y) & \text{if } T(j, y, z) \wedge \neg T(x, y, w), \text{ for } 0 \leq w < z \\ \psi(y) & \text{if } T(x, x, z) \wedge \neg T(j, y, w), \text{ for } 0 \leq w < z \\ \text{undefined} & \text{otherwise.} \end{cases}$$

Observe that if $x = y = j$, then $\Theta(j, j, z)$ is multiply defined, but since ψ extends φ, we get the same value $\psi(y) = \varphi(y)$, so Θ is a well defined partial function. Clearly, for all $(m, n) \in \mathbb{N}^2$, there is at most one $z \in \mathbb{N}$ so that $\Theta(x, y, z)$ is defined, so the function σ defined by

$$\sigma(x, y) = \begin{cases} z & \text{if } (x, y, z) \in \text{dom}(\Theta) \\ \text{undefined} & \text{otherwise} \end{cases}$$

is a partial computable function. Finally, let

$$\theta(x, y) = \Theta(x, y, \sigma(x, y)),$$

a partial computable function. It is easy to check that

$$\theta(x, y) = \begin{cases} \psi(y) & \text{if } x \in K, \\ \varphi(y) & \text{if } x \in \overline{K}, \end{cases}$$

for all $x, y \in \mathbb{N}$. By the s-m-n Theorem, there is a computable function h such that

$$\varphi_{h(x)}(y) = \theta(x, y)$$

for all $x, y \in \mathbb{N}$. $\square$

Observe that Proposition 7.8 yields a new proof that $\overline{\text{TOTAL}}$ is not listable (not c.e., not r.e.). Finally we can prove the extended Rice theorem.

Theorem 7.9. *(Extended Rice Theorem) The set P_C is listable (c.e., r.e.) iff there is a listable (c.e., r.e) set A such that*

$$\varphi_x \in C \quad \textit{iff} \quad \exists y \in A\, (P_y \subseteq \varphi_x).$$

Proof. Let $P_C = dom(\varphi_i)$. Using the s-m-n Theorem, there is a computable function k such that

$$\varphi_{k(y)} = P_y \quad \text{for all } y \in \mathbb{N}.$$

Define the listable set A such that

$$A = dom(\varphi_i \circ k).$$

Then

$$y \in A \quad \text{iff} \quad \varphi_i(k(y))\downarrow \quad \text{iff} \quad P_y \in C.$$

Next, using Proposition 7.7 and Proposition 7.8, it is easy to see that

$$\varphi_x \in C \quad \text{iff} \quad \exists y \in A\, (P_y \subseteq \varphi_x).$$

Indeed, if $\varphi_x \in C$, by Proposition 7.7, there is a finite subfunction $P_y \subseteq \varphi_x$ such that $P_y \in C$, but

$$P_y \in C \quad \text{iff} \quad y \in A,$$

as desired. On the other hand, if

$$P_y \subseteq \varphi_x$$

for some $y \in A$, then

$$P_y \in C,$$

and by Proposition 7.8, since φ_x extends P_y, we get

$$\varphi_x \in C. \qquad \square$$

7.3 Creative and Productive Sets; Incompleteness in Arithmetic

In this section, we discuss some special sets that have important applications in logic: creative and productive sets. These notions were introduced by Post and Dekker (1944, 1955). The concepts to be described are illustrated by the following situation. Assume that

$$W_x \subseteq \overline{K}$$

for some $x \in \mathbb{N}$ (recall that W_x was introduced in Definition 5.6). We claim that

$$x \in \overline{K} - W_x.$$

Indeed, if $x \in W_x$, then $\varphi_x(x)$ is defined, and by definition of K, we get $x \notin \overline{K}$, a contradiction. Therefore, $\varphi_x(x)$ must be undefined, that is,

$$x \in \overline{K} - W_x.$$

The above situation can be generalized as follows.

Definition 7.1. A set $A \subseteq \mathbb{N}$ is *productive* iff there is a total computable function f such that for every listable set W_x,

$$\text{if} \quad W_x \subseteq A \quad \text{then} \quad f(x) \in A - W_x \quad \text{for all } x \in \mathbb{N}.$$

The function f is called the *productive function of* A. A set A is *creative* if it is listable (c.e., r.e.) and if its complement $\overline{A}$ is productive.

As we just showed, K is creative and $\overline{K}$ is productive. It is also easy to see that TOTAL is productive. But TOTAL is worse than $\overline{K}$, because by Proposition 5.20, $\overline{\text{TOTAL}}$ is not listable.

The following facts are immediate consequences of the definition.

(1) A productive set is not listable (not c.e., not r.e.), since $A \neq W_x$ for all listable sets W_x (the image of the productive function f is a subset of $A - W_x$, which can't be empty since f is total).

(2) A creative set is not computable (not recursive).

Productiveness is a technical way of saying that a nonlistable set A is not listable in a rather strong and constructive sense. Indeed, there is a computable function f such that no matter how we attempt to approximate A with a listable set $W_x \subseteq A$, then $f(x)$ is an element in A not in W_x.

Remark: In Rogers [44] (Chapter 7, Section 3), the definition of a productive set only requires the productive function f to be *partial computable*. However, it is proven in Theorem XI of Rogers that this weaker requirement is equivalent to the stronger requirement of Definition 7.1.

Creative and productive sets arise in logic. The set of theorems of a logical theory is often creative. For example, the set of theorems in Peano's arithmetic is creative, and the set of true sentences of Peano's arithmetic is productive. This yields incompleteness results. We will return to this topic at the end of this section.

Proposition 7.10. *If a set A is productive, then it has an infinite listable (c.e., r.e.) subset.*

Proof. We first give an informal proof. Let f be the computable productive function of A. We define a computable function g as follows: Let x_0 be an index for the empty set, and let

$$g(0) = f(x_0).$$

Assuming that

$$\{g(0), g(1), \ldots, g(y)\}$$

is known, let x_{y+1} be an index for this finite set, and let

$$g(y+1) = f(x_{y+1}).$$

Since $W_{x_{y+1}} \subseteq A$, we have $f(x_{y+1}) \in A$.

For the formal proof, following Rogers [44] (Chapter 7, Section 7, Theorem X), we use the following facts whose proof is left as an exercise:

(1) There is a computable function u such that

$$W_{u(x,y)} = W_x \cup W_y.$$

(2) There is a computable function t such that

$$W_{t(x)} = \{x\}.$$

Letting x_0 be an index for the empty set, we define the function h as follows:

$$\begin{aligned} h(0) &= x_0, \\ h(y+1) &= u(t(f(y)), h(y)). \end{aligned}$$

We define g such that

$$g = f \circ h.$$

It is easily seen that g does the job. $\square$

Another important property of productive sets is the following.

Proposition 7.11. *If a set A is productive, then $\overline{K} \leq A$.*

Proof. Let f be a productive function for A. Using the s-m-n Theorem, we can find a computable function h such that

$$W_{h(y,x)} = \begin{cases} \{f(y)\} & \text{if } x \in K, \\ \emptyset & \text{if } x \in \overline{K}. \end{cases}$$

The above can be restated as follows:

$$\varphi_{h(y,x)}(z) = \begin{cases} 1 & \text{if } x \in K \text{ and } z = f(y), \\ \text{undefined} & \text{if } x \in \overline{K}, \end{cases}$$

for all $x, y, z \in \mathbb{N}$. By the third version of the recursion theorem (Theorem 7.3), there is a computable function g such that

$$W_{g(x)} = W_{h(g(x),x)} \quad \text{for all } x \in \mathbb{N}.$$

Let

$$k = f \circ g.$$

We claim that

$$x \in \overline{K} \quad \text{iff} \quad k(x) \in A \quad \text{for all } x \in \mathbb{N}.$$

Subtituting $g(x)$ for y in the equation for $W_{h(y,x)}$ and using the fact that $W_{g(x)} = W_{h(g(x),x)}$ and $k(x) = f(g(x))$, we get

$$W_{g(x)} = \begin{cases} \{f(g(x))\} = \{k(x)\} & \text{if } x \in K, \\ \emptyset & \text{if } x \in \overline{K}. \end{cases}$$

Because f is a productive function for A, if $x \in \overline{K}$, then $W_{g(x)} = \emptyset \subseteq A$, so $k(x) = f(g(x)) \in A$. Conversely, assume that $k(x) = f(g(x)) \in A$. If $x \in K$, then $W_{g(x)} = \{f(g(x))\}$, so $W_{g(x)} \subseteq A$, and since f is a productive function for A, we have $f(g(x)) \in A - W_{g(x)} = A - \{f(g(x))\}$, a contradiction. Therefore, $x \notin \overline{K}$ and the reduction is achieved. Thus, $\overline{K} \leq A$. ☐

Using Part (1) of Proposition 7.12 stated next we obtain the converse of Proposition 7.11. Thus a set A is productive iff $\overline{K} \leq A$. This fact is recorded in the next proposition.

The following results can also be shown.

Proposition 7.12. *The following facts hold.*

(1) If A is productive and $A \leq B$, then B is productive.

(2) A is creative iff A is complete.

(3) A is creative iff A is equivalent to K.

(4) A is productive iff $\overline{K} \leq A$.

Part (1) is easy to prove; see Rogers [44] (Chapter 7, Theorem V(b)). Part (2) is proven in Rogers [44] (Chapter 11, Corollary V). Part (3) follows from Part (2) since K is complete. Part (4) follows from Proposition 7.11 and Part (1).

We conclude with a discussion of the significance of the notions of productive and creative sets to logic. A more detailed discussion can be found in Rogers [44] (Chapter 7, Section 8). In Section 2.16 we discussed Peano arithmetic and the reader is invited to review it. It is convenient to add a countable set of constants $0, 1, 2, \ldots$, denoting the natural numbers to the language of arithmetic, and the new axioms

$$S^n(0) = n, \quad n \in \mathbb{N}.$$

By a now fairly routine process (using a pairing function and an extended pairing function), it is possible to assign a Gödel number $\#(A)$ to every first-order sentence A in the language of arithmetic; see Enderton [12] (Chapter III) or Kleene I.M. [32] (Chapter X). With some labor, it is possible to construct a formula F_x with one free variable x having the following property:

$$\begin{aligned}
&n \in K \ \text{ iff } \ (F_n \text{ is true in } \mathbb{N}) \\
&n \notin K \ \text{ iff } \ (F_n \text{ is false in } \mathbb{N}) \text{ iff } (\neg F_x \text{ is true in } \mathbb{N}).
\end{aligned}$$

One should not underestimate the technical difficulty of this task. One of Gödel's most original steps in proving his first incompleteness theorem was to define a variant of the formula F_x. Later on, simpler proofs were given, but they are still very technical. The brave reader should attempt to solve Exercises 7.64 and 7.65 in Rogers [44].

Observe that the sentences F_n are special kinds of sentences of arithmetic but of couse there are many more sentences of arithmetic. The following "basic lemma" from Rogers [44] (Chapter 7, Section 8) is easily shown.

Proposition 7.13. *For any two subsets S and T of $\mathbb{N}$, if T is listable and if $S \cap T$ is productive, then S is productive. In particular, if T is computable and if $S \cap T$ is productive, then S is productive.*

With a slight abuse of notation, we say that a set T is sentences of arithmetic is computable (*resp.* listable) iff the set of Gödel numbers $\#(A)$ of sentences A in T is computable (*resp.* listable). Then the following remarkable (historically shocking) facts hold.

Theorem 7.14. *(Unaxiomatizability of arithmetic) The following facts hold.*

(1) The set of sentences of arithmetic true in $\mathbb{N}$ is a productive set. Consequently, the set of true sentences is not listable.

(2) *The set of sentences of arithmetic false in* $\mathbb{N}$ *is a productive set. Consequently, the set of false sentences is not listable.*

Proof sketch. (1) It is easy to show that the set $\{\neg F_x \mid x \in \mathbb{N}\}$ is computable. Since

$$\{n \in \mathbb{N} \mid \neg F_n \text{ is true in } \mathbb{N}\} = \overline{K}$$

is productive and

$$\begin{aligned} \{A \mid A \text{ is true in } \mathbb{N}\} \cap \{\neg F_x \mid x \in \mathbb{N}\} &= \{\neg F_x \mid \neg F_x \text{ is true in } \mathbb{N}\} \\ &= \{\neg F_x \mid x \in \overline{K}\}, \end{aligned}$$

by Proposition 7.13, the set $\{A \mid A \text{ is true in } \mathbb{N}\}$ is also productive.

(2) It is also easy to show that the set $\{F_x \mid x \in \mathbb{N}\}$ is computable. Since

$$\{n \in \mathbb{N} \mid F_n \text{ is false in } \mathbb{N}\} = \overline{K}$$

is productive and

$$\begin{aligned} \{A \mid A \text{ is false in } \mathbb{N}\} \cap \{F_x \mid x \in \mathbb{N}\} &= \{F_x \mid F_x \text{ is false in } \mathbb{N}\} \\ &= \{F_x \mid x \in \overline{K}\}, \end{aligned}$$

by Proposition 7.13, the set $\{A \mid A \text{ is false in } \mathbb{N}\}$ is also productive. $\square$

Definition 7.2. A proof system for arithmetic is *axiomatizable* if the the set of provable sentences is listable.

Since the set of provable sentences of an axiomatizable proof system is listable, Theorem 7.14 annihilates any hope of finding an axiomatization of arithmetic. Theorem 7.14 also shows that it is impossible to decide effectively (algorithmically) whether a sentence of arithmetic is true. In fact the set of true sentences of arithmetic is *not even listable*.

If we consider proof systems for arithmetic, such as Peano arithmetic, then creative sets show up.

Definition 7.3. A proof system for arithmetic is *sound* if every provable sentence is true (in $\mathbb{N}$). A proof system is *consistent* if there is no sentence A such that both A and $\neg A$ are provable.

Clearly, a sound proof system is consistent.

Assume that a proof system for arithmetic is sound and strong enough so that the formula F_x with the free variable x introduced just before Proposition 7.13 has the following properties:

$$\begin{aligned} &n \in K \ \text{ iff } \ (F_n \text{ is provable}) \\ &n \notin K \ \text{ iff } \ (F_n \text{ is not provable}). \end{aligned}$$

Peano arithmetic is such a proof system. Then we have the following theorem.

Theorem 7.15. *(Undecidability of provablity in arithmetic) Consider any axiomatizable proof system for arithmetic satisfying the hypotheses stated before the statement of the theorem. The following facts hold.*

(1) The set of unprovable sentences of arithmetic is a productive set. Consequently, the set of unprovable sentences is not listable.

(2) The set of provable sentences of arithmetic is a creative set. Consequently, the set of provable sentences is not computable.

As a corollary of Theorem 7.15, there is no algorithm to decide whether a sentence of arithmetic is provable or not. But things are worse. Because the set of unprovable sentences of arithmetic is productive, there is a recursive function f, which for any attempt to find a listable subset W of the nonprovable sentences of arithmetic, produces another nonprovable sentence not in W.

Theorem 7.15 also implies Gödel's first incompleteness theorem. Indeed, it is immediately seen that the set $\{F_x \mid \neg F_x \text{ is provable}\}$ is listable (because $\{\neg F_x \mid x \in \mathbb{N}\}$ is computable and $\{A \mid A \text{ is provable}\}$ is listable). But since our proof system is assumed to be sound, $\neg F_x$ provable implies that F_x is *not* provable, so by

$$n \notin K \text{ iff } (F_n \text{ is not provable}),$$

we have

$$\{x \in \mathbb{N} \mid \neg F_x \text{ is provable}\} \subseteq \{x \in \mathbb{N} \mid F_x \text{ is not provable}\} = \overline{K}.$$

Since $\overline{K}$ is productive and $\{x \in \mathbb{N} \mid \neg F_x \text{ is provable}\}$ is listable, we have

$$W_y = \{x \in \mathbb{N} \mid \neg F_x \text{ is provable}\}$$

for some y, and if f is the productive function associated with $\overline{K}$, then for $x_0 = f(y)$ we have

$$F_{x_0} \in \{F_x \mid F_x \text{ is not provable}\} - \{\neg F_x \mid \neg F_x \text{ is provable}\},$$

that is, *both* F_{x_0} *and* $\neg F_{x_0}$ are *not* provable. Furthermore, since

$$n \notin K \text{ iff } (F_n \text{ is not provable})$$

and

$$n \notin K \text{ iff } (F_n \text{ is false in } \mathbb{N})$$

we see that F_{x_0} is false in $\mathbb{N}$, and so $\neg F_{x_0}$ is true in $\mathbb{N}$. In summary, we proved the following result.

Theorem 7.16. *(Incompleteness in arithmetic (Gödel 1931)) Consider any axiomatizable proof system for arithmetic satisfying the hypotheses stated earlier. Then there exists a sentence F of arithmetic ($F = \neg F_{x_0}$) such that neither F nor $\neg F$ are provable. Furthermore, F is true in $\mathbb{N}$.*

Theorem 7.15 holds under the weaker assumption that the proof system is *consistent* (as opposed to sound), and that there is a formula G with one free variable x such that

$$n \in K \text{ iff } (G_n \text{ is provable}).$$

The formula G is due to Rosser. The incompleteness theorem (Theorem 7.16) also holds under the weaker assumption of consistency, except for the statement that F is true.

To summarize informally the above negative results:

1. No (effective) axiomatization of mathematics can exactly capture all true statements of arithmetic.

2. From any (effective) axiomatization which yields only true statements of arithmetic, a new true statement can be found *not provable* in that axiomatization.

Fact (2) is what inspired Post to use the term *creative* for the type of sets arising in Definition 7.1. Indeed, one has to be creative to capture truth in arithmetic.

Another (relatively painless) way to prove incompleteness results in arithmetic is to use Diophantine definability; see Section 8.4.

Chapter 8

Listable Sets and Diophantine Sets; Hilbert's Tenth Problem

8.1 Diophantine Equations and Hilbert's Tenth Problem

There is a deep and a priori unexpected connection between the theory of computable and listable sets and the solutions of polynomial equations involving polynomials in several variables with integer coefficients. These are polynomials in $n \geq 1$ variables $x_1, \ldots, x_n$ which are finite sums of *monomials* of the form

$$a x_1^{k_1} \cdots x_n^{k_n},$$

where $k_1, \ldots, k_n \in \mathbb{N}$ are nonnegative integers, and $a \in \mathbb{Z}$ is an integer (possibly negative). The natural number $k_1 + \cdots + k_n$ is called the *degree* of the monomial $a x_1^{k_1} \cdots x_n^{k_n}$.

For example, if $n = 3$, then

1. $5,\ -7$, are monomials of degree 0.
2. $3x_1,\ -2x_2$, are monomials of degree 1.
3. $x_1x_2,\ 2x_1^2,\ 3x_1x_3,\ -5x_2^2$, are monomials of degree 2.
4. $x_1x_2x_3,\ x_1^2x_3,\ -x_2^3$, are monomials of degree 3.
5. $x_1^4,\ -x_1^2x_3^2,\ x_1x_2^2x_3$, are monomials of degree 4.

It is convenient to introduce multi-indices, where an *n-dimensional multi-index* is an n-tuple $\alpha = (k_1, \ldots, k_n)$ with $n \geq 1$ and $k_i \in \mathbb{N}$. Let $|\alpha| = k_1 + \cdots + k_n$. Then we can write

$$x^\alpha = x_1^{k_1} \cdots x_n^{k_n}.$$

For example, for $n = 3$,

$$x^{(1,2,1)} = x_1 x_2^2 x_3,\ x^{(0,2,2)} = x_2^2 x_3^2.$$

Definition 8.1. A *polynomial* $P(x_1, \ldots, x_n)$ in the variables $x_1, \ldots, x_n$ with integer coefficients is a finite sum of monomials of the form

$$P(x_1, \ldots, x_n) = \sum_{\alpha} a_\alpha x^\alpha,$$

where the α's are n-dimensional multi-indices, and with $a_\alpha \in \mathbb{Z}$. The maximum of the degrees $|\alpha|$ of the monomials $a_\alpha x^\alpha$ is called the *total degree* of the polynomial $P(x_1, \ldots, x_n)$. The set of all such polynomials is denoted by $\mathbb{Z}[x_1, \ldots, x_n]$.

Sometimes, we write P instead of $P(x_1, \ldots, x_n)$. We also use variables x, y, z *etc.* instead of $x_1, x_2, x_3, \ldots$.

For example, $2x - 3y - 1$ is a polynomial of total degree 1, $x^2 + y^2 - z^2$ is a polynomial of total degree 2, and $x^3 + y^3 + z^3 - 29$ is a polynomial of total degree 3.

Mathematicians have been interested for a long time in the problem of solving equations of the form

$$P(x_1, \ldots, x_n) = 0,$$

with $P \in \mathbb{Z}[x_1, \ldots, x_n]$, seeking only *integer solutions* for $x_1, \ldots, x_n$.

Diophantus of Alexandria, a Greek mathematician of the 3rd century, was one of the first to investigate such equations. For this reason, seeking integer solutions of polynomials in $\mathbb{Z}[x_1, \ldots, x_n]$ is referred to as *solving Diophantine equations*.

This problem is not as simple as it looks. The equation

$$2x - 3y - 1 = 0$$

obviously has the solution $x = 2, y = 1$, and more generally $x = -1 + 3a$, $y = -1 + 2a$, for any integer $a \in \mathbb{Z}$.

The equation

$$x^2 + y^2 - z^2 = 0$$

has the solution $x = 3$, $y = 4$, $z = 5$, since $3^2 + 4^2 = 9 + 16 = 25 = 5^2$. More generally, the reader should check that

$$x = t^2 - 1,\ y = 2t,\ z = t^2 + 1$$

is a solution for all $t \in \mathbb{Z}$.

The equation

$$x^3 + y^3 + z^3 - 29 = 0$$

has the solution $x = 3$, $y = 1$, $z = 1$.

What about the equation

$$x^3 + y^3 + z^3 - 30 = 0?$$

Amazingly, the only known integer solution is

$$(x, y, z) = (-283059965, -2218888517, 2220422932),$$

discovered in 1999 by E. Pine, K. Yarbrough, W. Tarrant, and M. Beck, following an approach suggested by N. Elkies.

And what about solutions of the equation

$$x^3 + y^3 + z^3 - 33 = 0?$$

Well, nobody knows whether this equation is solvable in integers!

In 1900, at the International Congress of Mathematicians held in Paris, the famous mathematician David Hilbert presented a list of ten open mathematical problems. Soon after, Hilbert published a list of 23 problems. The tenth problem is this:

Hilbert's tenth problem (H10)

Find an algorithm that solves the following problem:

Given as input a polynomial $P \in \mathbb{Z}[x_1, \ldots, x_n]$ with integer coefficients, return YES or NO, according to whether there exist integers $a_1, \ldots, a_n \in \mathbb{Z}$ so that $P(a_1, \ldots, a_n) = 0$; that is, the Diophantine equation $P(x_1, \ldots, x_n) = 0$ has a solution.

It is important to note that at the time Hilbert proposed his tenth problem, a rigorous mathematical definition of the notion of algorithm did not exist. In fact, the machinery needed to even define the notion of algorithm did not exist. It is only around 1930 that precise definitions of the notion of computability due to Turing, Church, and Kleene, were formulated, and soon after shown to be all equivalent.

So to be precise, the above statement of Hilbert's tenth should say: find a RAM program (or equivalently a Turing machine) that solves the following problem: ...

In 1970, the following somewhat surprising resolution of Hilbert's tenth problem was reached:

Theorem (Davis-Putnam-Robinson-Matiyasevich)

Hilbert's tenth problem is undecidable; that is, there is no algorithm for solving Hilbert's tenth problem.

In 1962, Davis, Putnam and Robinson had shown that if a fact known as *Julia Robinson hypothesis* could be proven, then Hilbert's tenth problem would be undecidable. At the time, the Julia Robinson hypothesis seemed implausible to many, so it was a surprise when in 1970 Matiyasevich found a set satisfying the Julia Robinson hypothesis, thus completing the proof of the undecidability of Hilbert's tenth problem. It is also a bit startling that Matiyasevich' set involves the Fibonacci numbers.

A detailed account of the history of the proof of the undecidability of Hilbert's tenth problem can be found in Martin Davis' classical paper Davis [8].

Even though Hilbert's tenth problem turned out to have a negative solution, the knowledge gained in developing the methods to prove this result is very significant. What was revealed is that polynomials have considerable expressive powers. This is what we discuss in the next section.

8.2 Diophantine Sets and Listable Sets

We begin by showing that if we can prove that the version of Hilbert's tenth problem *with solutions restricted to belong to* $\mathbb{N}$ is undecidable, then Hilbert's tenth problem (with solutions in $\mathbb{Z}$ is undecidable).

Proposition 8.1. *If we had an algorithm for solving Hilbert's tenth problem (with solutions in $\mathbb{Z}$), then we would have an algorithm for solving Hilbert's tenth problem with solutions restricted to belong to $\mathbb{N}$ (that is, nonnegative integers).*

Proof. The above statement is not at all obvious, although its proof is short with the help of some number theory. Indeed, by a theorem of Lagrange (Lagrange's four square theorem), *every* natural number m can be represented as the sum of four squares,

$$m = a_0^2 + a_1^2 + a_2^2 + a_3^2, \quad a_0, a_1, a_2, a_3 \in \mathbb{Z}.$$

We reduce Hilbert's tenth problem restricted to solutions in $\mathbb{N}$ to Hilbert's tenth problem (with solutions in $\mathbb{Z}$). Given a Diophantine equation $P(x_1, \ldots, x_n) = 0$, we can form the polynomial

$$Q = P(u_1^2 + v_1^2 + y_1^2 + z_1^2, \ldots, u_n^2 + v_n^2 + y_n^2 + z_n^2)$$

in the $4n$ variables u_i, v_i, y_i, z_i ($1 \leq i \leq n$) obtained by replacing x_i by $u_i^2 + v_i^2 + y_i^2 + z_i^2$ for $i = 1, \ldots, n$. If $Q = 0$ has a solution $(p_1, q_1, r_1, s_1, \ldots, p_n, q_n, r_n, s_n,)$ with $p_i, q_i, r_i, s_i \in \mathbb{Z}$, then if we set $a_i = p_i^2 + q_i^2 + r_i^2 + s_i^2$, obviously $P(a_1, \ldots, a_n) = 0$ with $a_i \in \mathbb{N}$. Conversely, if $P(a_1, \ldots, a_n) = 0$ with $a_i \in \mathbb{N}$, then by Lagrange's theorem there exist some $p_i, q_i, r_i, s_i \in \mathbb{Z}$ (in fact $\mathbb{N}$) such that $a_i = p_i^2 + q_i^2 + r_i^2 + s_i^2$ for $i = 1, \ldots, n$, and the equation $Q = 0$ has the solution $(p_1, q_1, r_1, s_1, \ldots, p_n, q_n, r_n, s_n,)$ with $p_i, q_i, r_i, s_i \in \mathbb{Z}$. Therefore $Q = 0$ has a solution $(p_1, q_1, r_1, s_1, \ldots, p_n, q_n, r_n, s_n,)$ with $p_i, q_i, r_i, s_i \in \mathbb{Z}$ iff $P = 0$ has a solution $(a_1, \ldots, a_n)$ with $a_i \in \mathbb{N}$. If we had an algorithm to decide whether Q has a solution with its components in $\mathbb{Z}$, then we would have an algorithm to decide whether $P = 0$ has a solution with its components in $\mathbb{N}$. $\square$

As consequence, the contrapositive of Proposition 8.1 shows that if the version of Hilbert's tenth problem restricted to solutions in $\mathbb{N}$ is undecidable, so is Hilbert's original problem (with solutions in $\mathbb{Z}$).

In fact, the Davis-Putnam-Robinson-Matiyasevich theorem establishes the undecidability of the version of Hilbert's tenth problem restricted to solutions in $\mathbb{N}$. *From now on, we restrict our attention to this version of Hilbert's tenth problem.*

A key idea is to use Diophantine equations with parameters, to *define* sets of numbers.

For example, consider the polynomial

$$P_1(a, y, z) = (y + 2)(z + 2) - a.$$

For $a \in \mathbb{N}$ fixed, the equation $(y + 2)(z + 2) - a = 0$, equivalently

$$a = (y + 2)(z + 2),$$

has a solution with $y, z \in \mathbb{N}$ iff a is composite.

If we now consider the polynomial

$$P_2(a, y, z) = y(2z + 3) - a,$$

for $a \in \mathbb{N}$ fixed, the equation $y(2z + 3) - a = 0$, equivalently

$$a = y(2z + 3),$$

has a solution with $y, z \in \mathbb{N}$ iff a is not a power of 2.

For a slightly more complicated example, consider the polynomial

$$P_3(a, y) = 3y + 1 - a^2.$$

We leave it as an exercise to show that the natural numbers a that satisfy the equation $3y + 1 - a^2 = 0$, equivalently

$$a^2 = 3y + 1,$$

or $(a - 1)(a + 1) = 3y$, are of the form $a = 3k + 1$ or $a = 3k + 2$, for any $k \in \mathbb{N}$.

In the first case, if we let S_1 be the set of composite natural numbers, then we can write

$$S_1 = \{a \in \mathbb{N} \mid (\exists y, z)((y + 2)(z + 2) - a = 0)\},$$

where it is understood that the existentially quantified variables y, z take their values in $\mathbb{N}$.

In the second case, if we let S_2 be the set of natural numbers that are not powers of 2, then we can write

$$S_2 = \{a \in \mathbb{N} \mid (\exists y, z)(y(2z + 3) - a = 0)\}.$$

In the third case, if we let S_3 be the set of natural numbers that are congruent to 1 or 2 modulo 3, then we can write

$$S_3 = \{a \in \mathbb{N} \mid (\exists y)(3y + 1 - a^2 = 0)\}.$$

A more explicit Diophantine definition for S_3 is

$$S_3 = \{a \in \mathbb{N} \mid (\exists y)((a - 3y - 1)(a - 3y - 2) = 0)\}.$$

The natural generalization is as follows.

Definition 8.2. A set $S \subseteq \mathbb{N}$ of natural numbers is *Diophantine* (or *Diophantine definable*) if there is a polynomial $P(a, x_1, \ldots, x_n) \in \mathbb{Z}[a, x_1, \ldots, x_n]$, with $n \geq 0$[1] such that

$$S = \{a \in \mathbb{N} \mid (\exists x_1, \ldots, x_n)(P(a, x_1, \ldots, x_n) = 0)\},$$

where it is understood that the existentially quantified variables $x_1, \ldots, x_n$ take their values in $\mathbb{N}$. More generally, a relation $R \subseteq \mathbb{N}^m$ is *Diophantine* ($m \geq 2$) if there is a polynomial $P(a_1, \ldots, a_m, x_1, \ldots, x_n) \in \mathbb{Z}[a_1, \ldots, a_m, x_1, \ldots, x_n]$, with $n \geq 0$, such that

$$R = \{(a_1, \ldots, a_m) \in \mathbb{N}^m \mid (\exists x_1, \ldots, x_n)(P(a_1, \ldots, a_m, x_1, \ldots, x_n) = 0)\},$$

where it is understood that the existentially quantified variables $x_1, \ldots, x_n$ take their values in $\mathbb{N}$.

For example, the strict order relation $a_1 < a_2$ is defined as follows:

$$a_1 < a_2 \quad \text{iff} \quad (\exists x)(a_1 + 1 + x - a_2 = 0),$$

and the divisibility relation $a_1 \mid a_2$ (a_1 divides a_2) is defined as follows:

$$a_1 \mid a_2 \quad \text{iff} \quad (\exists x)(a_1 x - a_2 = 0).$$

What about the ternary relation $R \subseteq \mathbb{N}^3$ given by

$$(a_1, a_2, a_3) \in R \quad \text{if} \quad a_1 \mid a_2 \quad \text{and} \quad a_1 < a_3?$$

At first glance it is not obvious how to "convert" a conjunction of Diophantine definitions into a single Diophantine definition, but we can do this using the following trick: given any finite number of Diophantine equations in the variables $x_1, \ldots, x_n$,

$$P_1 = 0,\ P_2 = 0,\ \ldots,\ P_m = 0, \tag{$*$}$$

observe that $(*)$ has a solution $(a_1, \ldots, a_n)$, which means that $P_i(a_1, \ldots, a_n) = 0$ for $i = 1, \ldots, m$, iff the single equation

$$P_1^2 + P_2^2 + \cdots + P_m^2 = 0 \tag{$**$}$$

also has the solution $(a_1, \ldots, a_n)$. This is because, since $P_1^2, P_2^2, \ldots, P_m^2$ are all nonnegative, their sum is equal to zero iff they are *all equal to zero*, that is $P_i^2 = 0$ for $i = 1 \ldots, m$, which is equivalent to $P_i = 0$ for $i = 1 \ldots, m$.

Using this trick, we see that

$$(a_1, a_2, a_3) \in R \quad \text{iff} \quad (\exists u, v)((a_1 u - a_2)^2 + (a_1 + 1 + v - a_3)^2 = 0).$$

We can also define the notion of Diophantine function.

[1]We have to allow $n = 0$. Otherwise singleton sets would not be Diophantine.

Definition 8.3. A function $f\colon \mathbb{N}^n \to \mathbb{N}$ is *Diophantine* iff its graph $\{(a_1, \ldots, a_n, b) \subseteq \mathbb{N}^{n+1} \mid b = f(a_1, \ldots, a_n)\}$ is Diophantine.

For example, the pairing function J and the projection functions K, L due to Cantor introduced in Section 4.1 are Diophantine, since

$$\begin{aligned} z = J(x, y) \quad &\text{iff} \quad (x + y - 1)(x + y) + 2x - 2z = 0 \\ x = K(z) \quad &\text{iff} \quad (\exists y)((x + y - 1)(x + y) + 2x - 2z = 0) \\ y = L(z) \quad &\text{iff} \quad (\exists x)((x + y - 1)(x + y) + 2x - 2z = 0). \end{aligned}$$

How extensive is the family of Diophantine sets? The remarkable fact proved by Davis-Putnam-Robinson-Matiyasevich is that they coincide with the listable sets (the recursively enumerable sets). This is a highly nontrivial result.

The easy direction is the following result.

Proposition 8.2. *Every Diophantine set is listable (recursively enumerable).*

Proof sketch. Suppose S is given as

$$S = \{a \in \mathbb{N} \mid (\exists x_1, \ldots, x_n)(P(a, x_1, \ldots, x_n) = 0)\},$$

Using the extended pairing function $\langle x_1, \ldots, x_n\rangle_n$ of Section 4.1, we enumerate all n-tuples $(x_1, \ldots, x_n) \in \mathbb{N}^n$, and during this process we compute $P(a, x_1, \ldots, x_n)$. If $P(a, x_1, \ldots, x_n)$ is zero, then we output a, else we go on. This way, S is the range of a computable function, and it is listable. $\square$

It is also easy to see that every Diophantine function is partial computable. The main theorem of the theory of Diophantine sets is the following deep result.

Theorem 8.3. *(Davis-Putnam-Robinson-Matiyasevich, 1970) Every listable subset of* $\mathbb{N}$ *is Diophantine. Every partial computable function is Diophantine.*

Theorem 8.3 is often referred to as the *DPRM theorem.* A complete proof of Theorem 8.3 is provided in Davis [8]. As noted by Davis, although the proof is certainly long and nontrivial, it only uses elementary facts of number theory, nothing more sophisticated than the Chinese remainder theorem. Nevetherless, the proof is a tour de force.

One of the most difficult steps is to show that the exponential function $h(n, k) = n^k$ is Diophantine. This is done using the Pell equation. According to Martin Davis, the proof given in Davis [8] uses a combination of ideas from Matiyasevich and Julia Robinson. Matiyasevich's proof used the Fibonacci numbers.

Using some results from the theory of computation it is now easy to deduce that Hilbert's tenth problem is undecidable. To achieve this, recall that there are listable sets that are not computable. For example, it is shown in Lemma 5.11 that $K = \{x \in \mathbb{N} \mid \varphi_x(x) \text{ is defined}\}$ is listable but not computable. Since K is listable, by Theorem 8.3, it is defined by some Diophantine equation

$$P(a, x_1, \ldots, x_n) = 0,$$

which means that

$$K = \{a \in \mathbb{N} \mid (\exists x_1 \ldots, x_n)(P(a, x_1, \ldots, x_n) = 0)\}.$$

We have the following strong form of the undecidability of Hilbert's tenth problem, in the sense that it shows that Hilbert's tenth problem is already undecidable for a fixed Diophantine equation in one parameter.

Theorem 8.4. *There is no algorithm which takes as input the polynomial $P(a, x_1, \ldots, x_n)$ defining K and any natural number $a \in \mathbb{N}$ and decides whether*

$$P(a, x_1, \ldots, x_n) = 0.$$

Consequently, Hilbert's tenth problem is undecidable.

Proof. If there was such an algorithm, then K would be decidable, a contradiction.

Any algorithm for solving Hilbert's tenth problem could be used to decide whether or not $P(a, x_1, \ldots, x_n) = 0$, but we just showed that there is no such algorithm. $\square$

It is an open problem whether Hilbert's tenth problem is undecidable if we allow *rational solutions* (that is, $x_1, \ldots, x_n \in \mathbb{Q}$).

Alexandra Shlapentokh proved that various extensions of Hilbert's tenth problem are undecidable. These results deal with some algebraic number theory beyond the scope of these notes. Incidentally, Alexandra was an undergraduate at Penn, and she worked on a logic project for me (finding a Gentzen system for a subset of temporal logic).

Having now settled once and for all the undecidability of Hilbert's tenth problem, we now briefly explore some interesting consequences of Theorem 8.3.

8.3 Some Applications of the DPRM Theorem

The first application of the DRPM theorem is a particularly striking way of defining the listable subsets of $\mathbb{N}$ as the nonnegative ranges of polynomials with integer coefficients. This result is due to Hilary Putnam.

Theorem 8.5. *For every listable subset S of $\mathbb{N}$, there is some polynomial $Q(x, x_1, \ldots, x_n)$ with integer coefficients such that*

$$S = \{Q(a, b_1, \ldots, b_n) \mid Q(a, b_1, \ldots, b_n) \in \mathbb{N},\ a, b_1, \ldots, b_n \in \mathbb{N}\}.$$

Proof. By the DPRM theorem (Theorem 8.3), there is some polynomial $P(x, x_1, \ldots, x_n)$ with integer coefficients such that

$$S = \{a \in \mathbb{N} \mid (\exists x_1, \ldots, x_n)(P(a, x_1, \ldots, x_n) = 0)\}.$$

Let $Q(x, x_1, \ldots, x_n)$ be given by

$$Q(x, x_1, \ldots, x_n) = (x + 1)(1 - P^2(x, x_1, \ldots, x_n)) - 1.$$

We claim that Q satisfies the statement of the theorem. If $a \in S$, then $P(a, b_1, \ldots, b_n) = 0$ for some $b_1, \ldots, b_n \in \mathbb{N}$, so

$$Q(a, b_1, \ldots, b_n) = (a + 1)(1 - 0) - 1 = a.$$

This shows that all $a \in S$ show up the the nonnegative range of Q. Conversely, assume that $Q(a, b_1, \ldots, b_n) \geq 0$ for some $a, b_1, \ldots, b_n \in \mathbb{N}$. Then by definition of Q we must have

$$(a + 1)(1 - P^2(a, b_1, \ldots, b_n)) - 1 \geq 0,$$

that is,

$$(a + 1)(1 - P^2(a, b_1, \ldots, b_n)) \geq 1,$$

and since $a \in \mathbb{N}$, this implies that $P^2(a, b_1, \ldots, b_n) < 1$, but since P is a polynomial with integer coefficients and $a, b_1, \ldots, b_n \in \mathbb{N}$, the expression $P^2(a, b_1, \ldots, b_n)$ must be a nonnegative integer, so we must have

$$P(a, b_1, \ldots, b_n) = 0,$$

which shows that $a \in S$. $\square$

Remark: It should be noted that in general, the polynomials Q arising in Theorem 8.5 may take on negative integer values, and to obtain all listable sets, we must restrict ourself to their nonnegative range.

As an example, the set S_3 of natural numbers that are congruent to 1 or 2 modulo 3 is given by

$$S_3 = \{a \in \mathbb{N} \mid (\exists y)(3y + 1 - a^2 = 0)\}.$$

so by Theorem 8.5, S_3 is the nonnegative range of the polynomial

$$\begin{aligned} Q(x, y) &= (x + 1)(1 - (3y + 1 - x^2)^2)) - 1 \\ &= -(x + 1)((3y - x^2)^2 + 2(3y - x^2))) - 1 \\ &= (x + 1)(x^2 - 3y)(2 - (x^2 - 3y)) - 1. \end{aligned}$$

Observe that $Q(x, y)$ takes on negative values. For example, $Q(0, 0) = -1$. Also, in order for $Q(x, y)$ to be nonnegative, $(x^2 - 3y)(2 - (x^2 - 3y))$ must be positive, but this can only happen if $x^2 - 3y = 1$, that is, $x^2 = 3y + 1$, which is the original equation defining S_3.

There is no miracle. The nonnegativity of $Q(x, x_1, \ldots, x_n)$ must subsume the solvability of the equation $P(x, x_1, \ldots, x_n) = 0$.

A particularly interesting listable set is the set of primes. By Theorem 8.5, in theory, the set of primes is the positive range of some polynomial with integer coefficients.

Remarkably, some explicit polynomials have been found. This is a nontrivial task. In particular, the process involves showing that the exponential function is definable, which was the stumbling block to the completion of the DPRM theorem for many years.

To give the reader an idea of how the proof begins, observe by the Bezout identity, if $p = s + 1$ and $q = s!$, then we can assert that p and q are relatively prime ($\gcd(p, q) = 1$) as the fact that the Diophantine equation

$$ap - bq = 1$$

is satisfied for some $a, b \in \mathbb{N}$. Then it is not hard to see that $p \in \mathbb{N}$ is prime iff the following set of equations has a solution for $a, b, s, r, q \in \mathbb{N}$:

$$\begin{aligned}
p &= s + 1 \\
p &= r + 2 \\
q &= s! \\
ap - bq &= 1.
\end{aligned}$$

The problem with the above is that the equation $q = s!$ is not Diophantine. The next step is to show that the factorial function is Diophantine, and this involves a lot of work. One way to proceed is to show that the above system is equivalent to a system allowing the use of the exponential function. The final step is to show that the exponential function can be eliminated in favor of polynomial equations.

We refer the interested reader to the remarkable expository paper by Davis, Matiyasevich and Robinson [9] for details. Here is a polynomial of total degree 25 in 26 variables (due to J. Jones, D. Sato, H. Wada, D. Wiens) which produces the primes as its positive range:

$$\begin{aligned}
&(k+2)\Big[1 - ([wz + h + j - q]^2 + [(gk + 2g + k + 1)(h + j) + h - z]^2 \\
&+ [16(k+1)^3(k+2)(n+1)^2 + 1 - f^2]^2 \\
&+ [2n + p + q + z - e]^2 + [e^3(e+2)(a+1)^2 + 1 - o^2]^2 \\
&+ [(a^2 - 1)y^2 + 1 - x^2]^2 + [16r^2y^4(a^2 - 1) + 1 - u^2]^2 \\
&+ [((a + u^2(u^2 - a))^2 - 1)(n + 4dy)^2 + 1 - (x + cu)^2]^2 \\
&+ [(a^2 - 1)l^2 + 1 - m^2]^2 + [ai + k + 1 - l - i]^2 + [n + l + v - y]^2 \\
&+ [p + l(a - n - 1) + b(2an + 2a - n^2 - 2n - 2) - m]^2 \\
&+ [q + y(a - p - 1) + s(2ap + 2a - p^2 - 2p - 2) - x]^2 \\
&+ [z + pl(a - p) + t(2ap - p^2 - 1) - pm]^2)\Big].
\end{aligned}$$

Around 2004, Nachi Gupta, an undergraduate student at Penn, and I tried to produce the prime 2 as one of the values of the positive range of the above polynomial. It turns out that this leads to values of the variables that are so large that we never succeeded!

Other interesting applications of the DPRM theorem are the re-statements of famous open problems, such as the Riemann hypothesis, as the unsolvability of certain Diophantine equations. For all this, see Davis, Matiyasevich and Robinson [9]. One may also obtain a nice variant of Gödel's incompleteness theorem.

8.4 Gödel's Incompleteness Theorem

Gödel published his famous incompleteness theorem in 1931. At the time, his result rocked the mathematical world, and certainly the community of logicians.

In order to understand why his result had such impact one needs to step back in time. In the late 1800's, Hilbert had advanced the thesis that it should be possible to completely formalize mathematics in such a way that every true statement should be provable "mechanically." In modern terminology, Hilbert believed that one could design a theorem prover that should be complete. His quest is known as *Hilbert's program*. In order to achieve his goal, Hilbert was led to investigate the notion of proof, and with some collaborators including Ackerman, Hilbert developed a significant amount of what is known as *proof theory*. When the young Gödel announced his incompleteness theorem, Hilbert's program came to an abrupt halt. Even the quest for a complete proof system for arithmetic was impossible.

It should be noted that when Gödel proved his incompleteness theorem, computability theory basically did not exist, so Gödel had to start from scratch. His proof is really a tour de force. Gödel's theorem also triggered extensive research on the notion of computability and undecidability between 1931 and 1936, the major players being Church, Gödel himself, Herbrand, Kleene, Rosser, Turing, and Post.

In this section we will give a (deceptively) short proof that relies on the DPRM and the existence of universal functions. The proof is short because the hard work lies in the proof of the DPRM!

The first step is to translate the fact that there is a universal partial computable function φ_{univ} (see Proposition 4.6), such that for all $x, y \in \mathbb{N}$, if φ_x is the xth partial computable function, then

$$\varphi_x(y) = \varphi_{univ}(x, y).$$

Also recall from Definition 5.6 that for any acceptable indexing of the partial computable functions, the listable (c.e. r.e.) sets W_x are given by

$$W_x = dom(\varphi_x), \quad x \in \mathbb{N}.$$

Since φ_{univ} is a partial computable function, it can be converted into a Diophantine equation so that we have the following result.

Theorem 8.6. *(Universal Equation Theorem) There is a Diophantine equation $U(m, a, x_1, \ldots x_\nu) = 0$ such that for every listable (c.e., r.e.) set W_m ($m \in \mathbb{N}$) we have*

$$a \in W_m \quad \textit{iff} \quad (\exists x_1, \ldots, x_\nu)(U(m, a, x_1, \ldots, x_\nu) = 0).$$

Proof. We have

$$W_m = \{a \in \mathbb{N} \mid (\exists x_1)(\varphi_{univ}(m, a) = x_1)\},$$

and since φ_{univ} is partial computable, by the DPRM (Theorem 8.3), there is Diophantine polynomial $U(m, a, x_1, \ldots, x_\nu)$ such that

$$x_1 = \varphi_{univ}(m, a) \quad \text{iff} \quad (\exists x_2, \ldots, x_\nu)(U(m, a, x_1, \ldots, x_\nu) = 0),$$

and so

$$W_m = \{a \in \mathbb{N} \mid (\exists x_1, \ldots, x_\nu)(U(m, a, x_1, \ldots, x_\nu) = 0)\},$$

as claimed. $\square$

The Diophantine equation $U(m, a, x_1, \ldots x_\nu) = 0$ is called a *universal Diophantine equation*. It is customary to denote $U(m, a, x_1, \ldots x_\nu)$ by $P_m(a, x_1, \ldots, x_\nu)$.

Gödel's incompleteness theorem applies to sets of logical (first-order) formulae of arithmetic built from the mathematical symbols $0, S, +, \cdot, <$ and the logical connectives $\wedge, \vee, \neg, \Rightarrow, =, \forall, \exists$. Recall that logical equivalence, $\equiv$, is defined by

$$P \equiv Q \quad \text{iff} \quad (P \Rightarrow Q) \wedge (Q \Rightarrow P).$$

The term

$$\underbrace{S(S(\cdots(S}_{n}(0))\cdots))$$

is denoted by $S^n(0)$, and represents the natural number n.

For example,

$$\exists x(S(S(S(0))) < (S(S(0)) + x)),$$

$$\exists x \exists y \exists z((x > 0) \wedge (y > 0) \wedge (z > 0) \wedge ((x \cdot x + y \cdot y) = z \cdot z)),$$

and

$$\forall x \forall y \forall z((x > 0) \wedge (y > 0) \wedge (z > 0) \Rightarrow \neg((x \cdot x \cdot x \cdot x + y \cdot y \cdot y \cdot y) = z \cdot z \cdot z \cdot z))$$

are formulae in the language of arithmetic. All three are true. The first formula is satisfied by $x = S(S(0))$, the second by $x = S^3(0), y = S^4(0)$ and $z = S^5(0)$ (since $3^2 + 4^2 = 9 + 16 = 25 = 5^2$), and the third formula asserts a special case of Fermat's famous theorem: for every $n \geq 3$, the equation $x^n + y^n = z^n$ has *no solution* with $x, y, z \in \mathbb{N}$ and $x > 0, y > 0, z > 0$. The third formula corrresponds to $n = 4$. Even for this case, the proof is hard.

To be completely rigorous we should explain precisely what is a formal proof. Roughly speaking, a proof system consists of axioms and inference rule. A proof is a certain kind of tree whose nodes are labeled with formulae, and this tree is constructed in such a way that for every node some inference rule is applied. Proof systems are discussed in Chapter 1 and in more detail in Chapter 2. The reader is invited to review this material. Such proof systems are also presented in Gallier [19, 18].

Given a polynomial $P(x_1, \ldots, x_m)$ in $\mathbb{Z}[x_1, \ldots, x_m]$, we need a way to "prove" that some natural numbers $n_1, \ldots, n_m \in \mathbb{N}$ are a solution of the Diophantine equation

$$P(x_1, \ldots, x_m) = 0,$$

which means that we need to have enough formulae of arithmetic to allow us to simplify the expression $P(n_1, \ldots, n_m)$ and check whether or not it is equal to zero.

For example, if $P(x, y) = 2x - 3y - 1$, we have the solution $x = 2$ and $y = 1$. What we do is to group all monomials with positive signs, $2x$, and all monomials with negative signs, $3y + 1$, plug in the values for x and y, simplify using the arithmetic tables for $+$ and $\cdot$, and then compare the results. If they are equal, then we proved that the equation has a solution.

In our language, $x = S^2(0)$, $2x = S^2(0) \cdot x$, and $y = S^1(0)$, $3y + 1 = S^3(0) \cdot y + S(0)$. We need to simplify the expressions

$$2x = S^2(0) \cdot S^2(0) \quad \text{and} \quad 3y + 1 = S^3(0) \cdot S(0) + S(0).$$

Using the formulae

$$\begin{aligned} S^m(0) + S^n(0) &= S^{m+n}(0) \\ S^m(0) \cdot S^n(0) &= S^{mn}(0) \\ S^m(0) < S^n(0) &\quad \text{iff} \quad m < n, \end{aligned}$$

with $m, n \in \mathbb{N}$, we simplify $S^2(0) \cdot S^2(0)$ to $S^4(0)$, $S^3(0) \cdot S(0) + S(0)$ to $S^4(0)$, and we see that the results are equal.

In general, given a polynomial $P(x_1, \ldots, x_m)$ in $\mathbb{Z}[x_1, \ldots, x_m]$, we write it as

$$P(x_1, \ldots, x_m) = P_{\mathrm{pos}}(x_1, \ldots, x_m) - P_{\mathrm{neg}}(x_1, \ldots, x_m),$$

where $P_{\mathrm{pos}}(x_1, \ldots, x_m)$ consists of the monomials with positive coefficients, and $-P_{\mathrm{neg}}(x_1, \ldots, x_m)$ consists of the monomials with negative coefficients. Next we plug in $S^{n_1}(0), \ldots, S^{n_m}(0)$ in $P_{\mathrm{pos}}(x_1, \ldots, x_m)$, and evaluate using the formulae for the addition and multiplication tables obtaining a term of the form $S^p(0)$. Similarly, we plug in $S^{n_1}(0), \ldots, S^{n_m}(0)$ in $P_{\mathrm{neg}}(x_1, \ldots, x_m)$, and evaluate using the formulae for the addition and multiplication tables obtaining a term of the form $S^q(0)$. Then we test in parallel (using the above formulae) whether

$$S^p(0) = S^q(0), \quad \text{or} \quad S^p(0) < S^q(0), \quad \text{or} \quad S^q(0) < S^p(0).$$

Only one outcome is possible, and we obtain a proof that either $P(n_1, \ldots, n_m) = 0$ or $P(n_1, \ldots, n_m) \neq 0$.

A more economical way that does use not an infinite number of formulae expressing the addition and multiplication tables is to use various axiomatizations of arithmetic.

One axiomatization known as *Robinson arithmetic* (R. M. Robinson (1950)) consists of the following seven axioms:

$$\begin{aligned}
&\forall x \neg (S(x) = 0)\\
&\forall x \forall y ((S(x) = S(y)) \Rightarrow (x = y))\\
&\forall y ((y = 0) \vee \exists x (S(x) = y))\\
&\forall x (x + 0 = x)\\
&\forall x \forall y (x + S(y) = S(x + y))\\
&\forall x (x \cdot 0 = 0)\\
&\forall x \forall y (x \cdot S(y) = x \cdot y + x).
\end{aligned}$$

Peano arithmetic is obtained from Robinson arithmetic by adding a rule schema expressing induction:

$$[\varphi(0) \wedge \forall n (\varphi(n) \Rightarrow \varphi(n+1))] \Rightarrow \forall m \varphi(m),$$

where $\varphi(x)$ is any (first-order) formula of arithmetic. To deal with $<$, we also have the axiom

$$\forall x \forall y (x < y \equiv \exists z (S(z) + x = y)).$$

It is easy to prove that the formulae

$$\begin{aligned}
S^m(0) + S^n(0) &= S^{m+n}(0)\\
S^m(0) \cdot S^n(0) &= S^{mn}(0)\\
S^m(0) &< S^n(0) \quad \text{iff} \quad m < n,
\end{aligned}$$

are provable in Robinson arithmetic, and thus in Peano arithmetic (with $m, n \in \mathbb{N}$).

Gödel's incompleteness applies to sets $\mathcal{A}$ of formulae of arithmetic that are "nice" and strong enough. A set $\mathcal{A}$ of formulae is nice if it is listable and *consistent* (see Definition 7.3), which means that it is impossible to prove φ and $\neg\varphi$ from $\mathcal{A}$ for some formula φ. In other words, $\mathcal{A}$ is free of contradictions.

Since the axioms of Peano arithmetic are obviously true statements about $\mathbb{N}$ and since the induction principle holds for $\mathbb{N}$, the set of all formulae provable in Robinson arithmetic and in Peano arithmetic is consistent.

As in Section 7.3, it is possible to assign a Gödel number $\#(A)$ to every first-order sentence A in the language of arithmetic; see Enderton [12] (Chapter III) or Kleene I.M. [32] (Chapter X). With a slight abuse of notation, we say that a set T is sentences of arithmetic is computable (*resp.* listable) iff the set of Gödel numbers $\#(A)$ of sentences A in T is computable (*resp.* listable). It can be shown that the set of all formulae provable in Robinson arithmetic and in Peano arithmetic are listable.

Here is a rather strong version of Gödel's incompleteness from Davis, Matiyasevich and Robinson [9].

Theorem 8.7. *(Gödel's Incompleteness Theorem) Let $\mathcal{A}$ be a set of formulae of arithmetic satisfying the following properties:*

(a) The set $\mathcal{A}$ is consistent.

(b) The set $\mathcal{A}$ is listable (c.e., r.e.)

(c) The set $\mathcal{A}$ is strong enough to prove all formulae

$$\begin{aligned} S^m(0) + S^n(0) &= S^{m+n}(0) \\ S^m(0) \cdot S^n(0) &= S^{mn}(0) \\ S^m(0) &< S^n(0) \quad \text{iff} \quad m < n, \end{aligned}$$

for all $m, n \in \mathbb{N}$.

Then we can construct a Diophantine equation $F(x_1, \ldots, x_\nu) = 0$ corresponding to $\mathcal{A}$ such that $F(x_1, \ldots, x_\nu) = 0$ has **no** *solution with $x_1, \ldots, x_\nu \in \mathbb{N}$ but the formula*

$$\neg(\exists x_1, \ldots, x_\nu)(F(x_1, \ldots, x_\nu) = 0) \tag{$*$}$$

is **not** *provable from $\mathcal{A}$. In other words, there is a true statement of arithmetic not provable from $\mathcal{A}$; that is, $\mathcal{A}$ is incomplete.*

Proof. Define the subset $A \subseteq \mathbb{N}$ as follows:

$$A = \{a \in \mathbb{N} \mid \neg(\exists x_1, \ldots, x_\nu)(P_a(a, x_1, \ldots, x_\nu) = 0) \text{ is provable from } \mathcal{A}\}, \tag{$**$}$$

where $P_m(a, x_1, \ldots, x_\nu)$ is defined just after Theorem 8.6. Because by (b) $\mathcal{A}$ is listable, it is easy to see (because the set of formulae provable from a listable set is listable) that A is listable, so by the DPRM A is Diophantine, and by Theorem 8.6, there is some $k \in \mathbb{N}$ such that

$$A = W_k = \{a \in \mathbb{N} \mid (\exists x_1, \ldots, x_\nu)(P_k(a, x_1, \ldots, x_\nu) = 0).$$

The trick is now to see whether $k \in W_k$ or not.

We claim that $k \notin W_k$.

We proceed by contradiction. Assume that $k \in W_k$. This means that

$$(\exists x_1, \ldots, x_\nu)(P_k(k, x_1, \ldots, x_\nu) = 0), \tag{$\dagger_1$}$$

and since $A = W_k$, by $(**)$, that

$$\neg(\exists x_1, \ldots, x_\nu)(P_k(k, x_1, \ldots, x_\nu) = 0) \text{ is provable from } \mathcal{A}. \tag{$\dagger_2$}$$

By $(\dagger_1)$ and (c), since the equation $P_k(k, x_1, \ldots, x_\nu) = 0$ has a solution, we can prove the formula

$$(\exists x_1, \ldots, x_\nu)(P_k(k, x_1, \ldots, x_\nu) = 0)$$

from $\mathcal{A}$. By ($\dagger_2$), the formula $\neg(\exists x_1, \ldots, x_\nu)(P_k(k, x_1, \ldots, x_\nu) = 0)$ is provable from $\mathcal{A}$, but since $(\exists x_1, \ldots, x_\nu)(P_k(k, x_1, \ldots, x_\nu) = 0)$ is also provable from $\mathcal{A}$, this contradicts the fact that $\mathcal{A}$ is consistent (which is hypothesis (a)).

Therefore we must have $k \notin W_k$. This means that $P_k(k, x_1, \ldots, x_\nu) = 0$ has **no** solution with $x_1, \ldots, x_\nu \in \mathbb{N}$, and since $A = W_k$, the formula

$$\neg(\exists x_1, \ldots, x_\nu)(P_k(k, x_1, \ldots, x_\nu) = 0)$$

is **not** provable from $\mathcal{A}$. $\square$

Remark: Going back to the proof of Theorem 7.16, observe that A plays the role of $\{F_x \mid \neg F_x \text{ is provable}\}$, that k plays the role of x_0, and that the fact that

$$\neg(\exists x_1, \ldots, x_\nu)(P_k(k, x_1, \ldots, x_\nu) = 0)$$

is **not** provable from $\mathcal{A}$ corresponds to $\neg F_{x_0}$ being true.

As a corollary of Theorem 8.7, since the theorems provable in Robinson arithmetic satisfy (a), (b), (c), we deduce that there are true theorems of arithmetic not provable in Robinson arithmetic; in short, Robinson arithmetic is incomplete. Since Robinson arithmetic does not have induction axioms, this shows that induction is not the culprit behind incompleteness. Since Peano arithmetic is an extension (consistent) of Robinson arithmetic, it is also incomplete. This is Gödel's original incompleteness theorem, but Gödel had to develop from scratch the tools needed to prove his result, so his proof is very different (and a tour de force).

But the situation is even more dramatic. Adding a true unprovable statement to a set $\mathcal{A}$ satisfying (a), (b), (c) preserves properties (a), (b), (c), so there is no escape from incompleteness (unless perhaps we allow unreasonable sets of formulae violating (b)). The reader should compare this situation with the results given by Theorem 7.14 and Theorem 7.15.

Gödel's incomplenetess theorem is a negative result, in the sense that it shows that there is no hope of obtaining proof systems capable of proving all true statements for various mathematical theories such as arithmetic. We can also view Gödel's incomplenetess theorem positively as evidence that mathematicians will never be replaced by computers! There is always room for creativity.

The true but unprovable formulae arising in Gödel's incompleteness theorem are rather contrived and by no means "natural." For many years after Gödel's proof was published logicians looked for natural incompleteness phenomena. In the early 1980's such results were found, starting with a result of Kirby and Paris. Harvey Friedman then found more spectacular instances of natural incompleteness, one of which involves a finite miniaturization of Kruskal's tree theorem. The proof of such results uses some deep methods of proof theory involving a tool known as ordinal notations. A survey of such results can be found in Gallier [13].

Chapter 9

The Post Correspondence Problem; Applications to Undecidability Results

9.1 The Post Correspondence Problem

The Post correspondence problem (due to Emil Post) is another undecidable problem that turns out to be a very helpful tool for proving problems in logic or in formal language theory to be undecidable.

Definition 9.1. Let Σ be an alphabet with at least two letters. An instance of the *Post Correspondence problem* (for short, PCP) is given by two nonempty sequences $U = (u_1, \ldots, u_m)$ and $V = (v_1, \ldots, v_m)$ of strings $u_i, v_i \in \Sigma^*$. Equivalently, an instance of the PCP is a sequence of pairs $(u_1, v_1), \ldots, (u_m, v_m)$.

The problem is to find whether there is a (finite) sequence $(i_1, \ldots, i_p)$, with $i_j \in \{1, \ldots, m\}$ for $j = 1, \ldots, p$, so that

$$u_{i_1} u_{i_2} \cdots u_{i_p} = v_{i_1} v_{i_2} \cdots v_{i_p}.$$

Example 9.1. Consider the following problem:

$$(abab, ababaaa), (aaabbb, bb), (aab, baab), (ba, baa), (ab, ba), (aa, a).$$

There is a solution for the string 1234556:

$$abab\,aaabbb\,aab\,ba\,ab\,ab\,aa = ababaaa\,bb\,baab\,baa\,ba\,ba\,a.$$

If you are not convinced that this is a hard problem, try solving the following instance of the PCP:

$$\{(aab, a), (ab, abb), (ab, bab), (ba, aab).\}$$

The shortest solution is a sequence of length 66.

We are beginning to suspect that this is a hard problem. Indeed, it is undecidable!

Theorem 9.1. *(Emil Post, 1946) The Post correspondence problem is undecidable, provided that the alphabet Σ has at least two symbols.*

There are several ways of proving Theorem 9.1, but the strategy is more or less the same: reduce the halting problem to the PCP, by encoding sequences of ID's as partial solutions of the PCP. In Machtey and Young [36] (Section 2.6), the undecidability of the PCP is shown by demonstrating how to simulate the computation of a Turing machine as a sequence of ID's. We give a proof involving special kinds of RAM programs (called Post machines in Manna [37]), which is an adaptation of a proof due to Dana Scott presented in Manna [37] (Section 1.5.4, Theorem 1.8).

Proof. The first step of the proof is to show that a RAM program with $p \geq 2$ registers can be simulated by a RAM program using a single register. The main idea of the simulation is that by using the instructions `add`, `tail`, and `jmp`, it is possible to perform cyclic permutations on the string held by a register.

First we can also assume that RAM programs only uses instructions of the form

(1_j)	N		$\mathtt{add}_j$	X
(2)	N		`tail`	X
(6_ja)	N	X	$\mathtt{jmp}_j$	$N1a$
(6_jb)	N	X	$\mathtt{jmp}_j$	$N1b$
(7)	N		`continue`	

We can simulate $p \geq 2$ registers with a single register, by encoding the contents $r_1, \ldots, r_p$ of the p registers as the string

$$r_1\#r_2\#\cdots\#r_p,$$

using a single marker $\#$. For instance, if $p = 2$, the effect of the instruction $\mathtt{add}_b$ on register $R1$ is achieved as follows: Assuming that the initial contents are

$$aab\#baba$$

using cyclic permutations (also inserting or deleting $\#$ whenever necessary), we get

$$\begin{array}{ll}
aab\#baba & \\
ab\#baba\#a & \\
b\#baba\#aa & \\
baba\#aab & \\
baba\#aabb & \text{(add } b \text{ on the right)} \\
aba\#aabb\#b & \\
ba\#aabb\#ba & \\
a\#aabb\#bab & \\
aabb\#baba &
\end{array}$$

Similarly, the effect of the instruction `tail` on register $R2$ is achieved as follows

$$
\begin{aligned}
&aab\#baba\\
&ab\#baba\#a\\
&b\#baba\#aa\\
&baba\#aab\\
&aba\#aab \qquad \text{(delete the leftmost letter)}\\
&ba\#aab\#a\\
&a\#aab\#ab\\
&aab\#aba
\end{aligned}
$$

Since the halting problem for RAM programs is undecidable and since every RAM program can be simulated by another RAM with a single register, the halting problem for RAM programs with a single register is undecidable.

The second step of the proof is to reduce the halting problem for RAM programs with one register to the PCP (over an alphabet with at least two symbols).

Recall that $\Sigma = \{a_1, \ldots, a_k\}$. First, it is easily shown that every RAM program P (with a single register X) is equivalent to a RAM program P' such that all instructions are labeled with distinct line numbers, and such that there is only one occurrence of the instruction `continue` at the end of the program.

In order to obtain a reasonably simple reduction of the halting problem for RAM programs with a single register to the PCP, we modify the jump instructions as follows: the new instruction

$$\texttt{Jmp}\, N_1, \ldots, N_k, N_{k+1}$$

tests whether $head(X) = a_j$, with $1 \leq j \leq k$. Since there is a single register X, it is omitted in the instruction `Jmp`. If $head(X) = a_j$, then jump to the instruction labeled N_j *and* perform the `tail` instruction so that $X = tail(X)$, otherwise if $X = \epsilon$ (which implies that $j = k + 1$), jump to the instruction labeled N_{k+1}. The instruction `tail` is eliminated. We leave it as an exercise to show how to simulate the new instruction

$$\texttt{Jmp}\, N_1, \ldots, N_k, N_{k+1}$$

using the instructions `tail`, $\texttt{jmp}_j\, Na$ and $\texttt{jmp}_j\, Nb$, and vice-versa. From now on we will use the second version using the instructions

$$\texttt{Jmp}\, N_1, \ldots, N_k, N_{k+1}.$$

For the purpose of deciding whether a RAM program terminates, we may assume without loss of generality that we deal with programs that clear the register X when they halt. In fact, by adding an extra symbol # to the alphabet (which now has $k+1$ symbols), we may also assume that in every instruction

$$\texttt{Jmp}\, N_1, \ldots, N_{k+1}, N_{k+2},$$

N_{k+2} is the line number of the last instruction in the RAM program, which must be a `continue`. This implies that the program clears the register X before it halts. We can execute the instruction $\texttt{add}_{k+1}\ X$ at the very beginning of the program and perform an $\texttt{add}_{k+1}\ X$ after each `tail` instruction to make sure that in the new program the register X always has # as its rightmost symbol. When the original program performs an instruction

$$\texttt{Jmp}\, N_1, \ldots, N_{k+1}, N_{k+2}$$

with $X = \epsilon$, the new program performs the instruction

$$\texttt{Jmp}\, N_1, \ldots, N_{k+2}, N_1.$$

Since X is never empty during execution of the new program, the line number N_1 is irrelevant. Finally, when the original program halts, the new program clears the register X and then jumps to the last `continue`. We leave the details as an exercise.

From now on, we assume that $\Sigma = \{a_1, \ldots, a_k, \#\}$. Given a RAM program P satisfying all the restrictions described above, we construct an instance of the PCP as follows. Assume that P has q lines numbered $N1, \ldots, Nq$. The alphabet Δ of the PCP is $\Delta = \Sigma \cup \{*, N_0, N_1, \ldots, N_q\}$. Indeed, the construction requires one more line number N_0 to force a solution of the PCP to start with some specified pair.

The lists U and V are constructed so that given any nonempty input $x = x_1 \cdots x_m$ (with $x_i \in \Sigma$), the only possible U-lists u and V-lists v that could lead to a solution are of the form

$$u = N_0\, w_0 * N_1\, w_1 * \cdots * N_{i_{n-1}}\, w_{n-1} * N_{i_n}$$

and

$$v = N_0\, w_0 * N_1\, w_1 * \cdots * N_{i_{n-1}}\, w_{n-1} * N_{i_n}\, w_n * N_{i_{n+1}} *,$$

where each w_i is of the form

$$w_i = *\, w_{i,1} * \cdots * w_{i,n_i} \quad \text{or} \quad w_i = \epsilon,$$

with

$$w_0 = *\, x_1 * x_2 * \cdots * x_m,$$

where $w_{i,j} \in \Sigma$, $1 \leq j \leq n_i$, $1 \leq i \leq n$.

The sequence $N_1, \ldots, N_{i_{n+1}}$ is the sequence of line numbers of the instructions executed by the RAM program P after n steps, started on input x, and w_j is the value of the (single) register X just after executing the jth step, *i.e.*, the instruction at line number N_{i_j}. We make sure that the V-list is always ahead of the U-list by one instruction.

The lists U and V are defined according to the following rules. Rather than defining U and V explicitly, we define the pairs $(u_i,\, v_i)$, where $u_i \in U$ and $v_i \in V$.

To get started: we have the initial pair

$$(N_0,\; N_0 * x_1 * x_2 * \cdots * x_m * N_1 *).$$

To simulate an instruction

$$N_i \qquad \texttt{add}_j \quad X,$$

create the pair

$$(* \, N_i,\; a_j * N_{i+1} *), \quad \text{for all } a \in \Sigma.$$

To simulate an instruction of the form

$$N_i \qquad \texttt{Jmp}\, N_1, \ldots, N_{k+1}, N_{k+2},$$

create the pairs

$$(* \, N_i * \, a_j,\; N_j *), \quad 1 \leq j \leq k+1,$$

and

$$(* \, N_i * N_q, N_q).$$

To build up the register contents, we need pairs

$$(* \, a,\; a \, *), \quad \text{for all } a \in \Sigma.$$

Note that we used the alphabet $\Delta = \Sigma \cup \{*, N_0, N_1, \ldots, N_q\}$, which uses more than 2 symbols in general. Let us finish our reduction for an instance of the PCP over the alphabet Δ. Then after this construction is finished we will explain how to convert the instance of the PCP that we obtained to an instance of the PCP over a two-symbol alphabet.

The pairs of the PCP are designed so that the only possible U-lists u and V-lists v that could lead to a solution are of the form

$$u = N_0 \, w_0 * N_1 \, w_1 * \cdots * N_{i_{n-1}} \, w_{n-1} * N_{i_n}$$

and

$$v = N_0 \, w_0 * N_1 \, w_1 * \cdots * N_{i_{n-1}} \, w_{n-1} * N_{i_n} \, w_n * N_{i_{n+1}} *,$$

where each w_i is of the form

$$w_i = * \, w_{i,1} * \cdots * w_{i,n_i} \quad \text{or} \quad w_i = \epsilon,$$

with

$$w_0 = * \, x_1 * x_2 * \cdots * x_m,$$

where $w_{i,j} \in \Sigma$, $1 \leq j \leq n_i$, $1 \leq i \leq n$, and where v is an encoding of n steps of the computation of the RAM program P on input $x = x_1 \cdots x_m$, and u lags behind v by one step.

For example, let us see how the U-list and the V-list are updated, assuming that N_{i_n} is the following instruction:

N_{i_n} `add`$_b$ X

Just after execution of the instruction at line number N_{i_n}, we have

$$u = N_0\, w_0 * N_1\, w_1 * \cdots *\; N_{i_{n-1}}$$

and

$$v = N_0\, w_0 * N_1\, w_1 * \cdots * N_{i_{n-1}}\, w_{n-1} * N_{i_n}\; * .$$

Since $w_{n-1} = * \, w_{n-1,1} * \cdots * w_{n-1,n_{n-1}}$, using the pairs

$$(* \, w_{n-1,1},\, w_{n-1,1} *),\ (* \, w_{n-1,2},\, w_{n-1,2} *), \cdots, (* \, w_{n-1,n_{n-1}},\, w_{n-1,n_{n-1}} *),$$

we get

$$u = N_0\, w_0 * N_1\, w_1 * \cdots *\; N_{i_{n-1}}\, w_{n-1}$$

and

$$v = N_0\, w_0 * N_1\, w_1 * \cdots * N_{i_{n-1}}\, w_{n-1} * N_{i_n}\, w_{n-1}\; * .$$

Next we use the pair

$$(* \, N_{i_n},\, b * N_{i_{n+1}} *)$$

simulating `add`$_b$, and we get

$$u = N_0\, w_0 * N_1\, w_1 * \cdots *\; N_{i_{n-1}}\, w_{n-1} * N_{i_n}$$

and

$$v = N_0\, w_0 * N_1\, w_1 * \cdots * N_{i_{n-1}}\; w_{n-1} * N_{i_n}\, w_{n-1}\; * b * N_{i_{n+1}}\; * .$$

Observe that the only chance for getting a solution of the PCP is to start with the pairs involving N_0. It is easy to see that the PCP constructed from P has a solution iff P halts on input x. However, the halting problem for RAM's with a single register is undecidable, and thus, the PCP over the alphabet Δ is also undecidable.

It remains to show that we can recode the instance of the PCP that we obtained over the alphabet $\Delta = \Sigma \cup \{*, N_0, N_1, \ldots, N_q\}$ as an instance of the PCP over the alphabet $\{a_1, *\}$. To achieve this, we recode each symbol a_i in Σ as $*a_1^i$ (with $a_{k+1} = \#$) and each N_j as $*a_1^{k+j+2}$. This way, we are only using the alphabet $\Delta = \{a_1, *\}$. We need the second character $*$, whose purpose is to avoid trivial solutions of the form

$$(u, u).$$

This could happen if we had used pairs (a, a) to build up the register. Then we substitute $*a_1^i$ for a_i and $*a_1^{k+j+2}$ for N_j in the pairs that we created. Observe that the pairs $(*a, a*)$ become pairs involving longer strings. It is easy to see that the original PCP over Δ has a solution iff the new PCP over $\{a_1, *\}$ has a solution, so the PCP over two-letter alphabet is undecidable. $\square$

In the next two sections we present some undecidability results for context-free grammars and context-free languages.

9.2 Some Undecidability Results for CFG's

Theorem 9.2. *It is undecidable whether a context-free grammar is ambiguous.*

Proof. We reduce the PCP to the ambiguity problem for CFG's. Given any instance $U = (u_1, \ldots, u_m)$ and $V = (v_1, \ldots, v_m)$ of the PCP, let $c_1, \ldots, c_m$ be m new symbols, and consider the following languages:

$$\begin{aligned} L_U = \{u_{i_1} \cdots u_{i_p} c_{i_p} \cdots c_{i_1} \mid 1 \leq i_j \leq m, \\ 1 \leq j \leq p,\, p \geq 1\}, \\ L_V = \{v_{i_1} \cdots v_{i_p} c_{i_p} \cdots c_{i_1} \mid 1 \leq i_j \leq m, \\ 1 \leq j \leq p,\, p \geq 1\}, \end{aligned}$$

and $L_{U,V} = L_U \cup L_V$.

We can easily construct a CFG, $G_{U,V}$, generating $L_{U,V}$. The productions are:

$$\begin{aligned} S &\longrightarrow S_U \\ S &\longrightarrow S_V \\ S_U &\longrightarrow u_i S_U c_i \\ S_U &\longrightarrow u_i c_i \\ S_V &\longrightarrow v_i S_V c_i \\ S_V &\longrightarrow v_i c_i. \end{aligned}$$

It is easily seen that the PCP for (U, V) has a solution iff $L_U \cap L_V \neq \emptyset$ iff G is ambiguous. $\square$

Remark: As a corollary, we also obtain the following result: it is undecidable for arbitrary context-free grammars G_1 and G_2 whether $L(G_1) \cap L(G_2) = \emptyset$ (see also Theorem 9.4).

Recall that the computations of a Turing Machine, M, can be described in terms of instantaneous descriptions, *upav*.

We can encode computations

$$ID_0 \vdash ID_1 \vdash \cdots \vdash ID_n$$

halting in a proper ID, as the language, L_M, consisting all of strings

$$w_0\#w_1^R\#w_2\#w_3^R\# \cdots \#w_{2k}\#w_{2k+1}^R,$$

or

$$w_0\#w_1^R\#w_2\#w_3^R\# \cdots \#w_{2k-2}\#w_{2k-1}^R\#w_{2k},$$

where $k \geq 0$, w_0 is a starting ID, $w_i \vdash w_{i+1}$ for all i with $0 \leq i < 2k+1$ and w_{2k+1} is proper halting ID in the first case, $0 \leq i < 2k$ and w_{2k} is proper halting ID in the second case.

The language L_M turns out to be the intersection of two context-free languages L_M^0 and L_M^1 defined as follows:

(1) The strings in L_M^0 are of the form

$$w_0\#w_1^R\#w_2\#w_3^R\# \cdots \#w_{2k}\#w_{2k+1}^R$$

or

$$w_0\#w_1^R\#w_2\#w_3^R\# \cdots \#w_{2k-2}\#w_{2k-1}^R\#w_{2k},$$

where $w_{2i} \vdash w_{2i+1}$ for all $i \geq 0$, and w_{2k} is a proper halting ID in the second case.

(2) The strings in L_M^1 are of the form

$$w_0\#w_1^R\#w_2\#w_3^R\# \cdots \#w_{2k}\#w_{2k+1}^R$$

or

$$w_0\#w_1^R\#w_2\#w_3^R\# \cdots \#w_{2k-2}\#w_{2k-1}^R\#w_{2k},$$

where $w_{2i+1} \vdash w_{2i+2}$ for all $i \geq 0$, w_0 is a starting ID, and w_{2k+1} is a proper halting ID in the first case.

Theorem 9.3. *Given any Turing machine M, the languages L_M^0 and L_M^1 are context-free, and $L_M = L_M^0 \cap L_M^1$.*

Proof. We can construct PDA's accepting L_M^0 and L_M^1. It is easily checked that $L_M = L_M^0 \cap L_M^1$. $\square$

As a corollary, we obtain the following undecidability result:

Theorem 9.4. *It is undecidable for arbitrary context-free grammars G_1 and G_2 whether $L(G_1) \cap L(G_2) = \emptyset$.*

Proof. We can reduce the problem of deciding whether a partial recursive function is undefined everywhere to the above problem. By Rice's theorem, the first problem is undecidable.

However, this problem is equivalent to deciding whether a Turing machine never halts in a proper ID. By Theorem 9.3, the languages L_M^0 and L_M^1 are context-free. Thus, we can construct context-free grammars G_1 and G_2 so that $L_M^0 = L(G_1)$ and $L_M^1 = L(G_2)$. Then M never halts in a proper ID iff $L_M = \emptyset$ iff (by Theorem 9.3), $L_M = L(G_1) \cap L(G_2) = \emptyset$. $\square$

Given a Turing machine M, the language L_M is defined over the alphabet $\Delta = \Gamma \cup Q \cup \{\#\}$. The following fact is also useful to prove undecidability:

Theorem 9.5. *Given any Turing machine M, the language $\Delta^* - L_M$ is context-free.*

Proof. One can easily check that the conditions for not belonging to L_M can be checked by a PDA. $\square$

As a corollary, we obtain:

Theorem 9.6. *Given any context-free grammar, $G = (V, \Sigma, P, S)$, it is undecidable whether $L(G) = \Sigma^*$.*

Proof. We can reduce the problem of deciding whether a Turing machine never halts in a proper ID to the above problem.

Indeed, given M, by Theorem 9.5, the language $\Delta^* - L_M$ is context-free. Thus, there is a CFG, G, so that $L(G) = \Delta^* - L_M$. However, M never halts in a proper ID iff $L_M = \emptyset$ iff $L(G) = \Delta^*$. $\square$

As a consequence, we also obtain the following:

Theorem 9.7. *Given any two context-free grammar, G_1 and G_2, and any regular language, R, the following facts hold:*

(1) $L(G_1) = L(G_2)$ *is undecidable.*

(2) $L(G_1) \subseteq L(G_2)$ *is undecidable.*

(3) $L(G_1) = R$ *is undecidable.*

(4) $R \subseteq L(G_2)$ *is undecidable.*

In contrast to (4), the property $L(G_1) \subseteq R$ is decidable!

9.3 More Undecidable Properties of Languages; Greibach's Theorem

We discuss a nice theorem of S. Greibach, which is a sort of version of Rice's theorem for families of languages.

Let $\mathcal{L}$ be a countable family of languages. We assume that there is a coding function $c\colon \mathcal{L} \to \mathbb{N}$ and that this function can be extended to code the regular languages (all alphabets are subsets of some given countably infinite set).

We also assume that $\mathcal{L}$ is effectively closed under union, and concatenation with the regular languages.

This means that given any two languages L_1 and L_2 in $\mathcal{L}$, we have $L_1 \cup L_2 \in \mathcal{L}$, and $c(L_1 \cup L_2)$ is given by a recursive function of $c(L_1)$ and $c(L_2)$, and that for every regular language R, we have $L_1R \in \mathcal{L}$, $RL_1 \in \mathcal{L}$, and $c(RL_1)$ and $c(L_1R)$ are recursive functions of $c(R)$ and $c(L_1)$.

Given any language, $L \subseteq \Sigma^*$, and any string, $w \in \Sigma^*$, we define L/w by

$$L/w = \{u \in \Sigma^* \mid uw \in L\}.$$

Theorem 9.8. *(Greibach) Let $\mathcal{L}$ be a countable family of languages that is effectively closed under union and concatenation with the regular languages, and assume that the problem $L = \Sigma^*$ is undecidable for $L \in \mathcal{L}$ and any given sufficiently large alphabet Σ. Let P be any nontrivial property of languages that is true for the regular languages, so that if $P(L)$ holds for any $L \in \mathcal{L}$, then $P(L/a)$ also holds for any letter a. Then P is undecidable for $\mathcal{L}$.*

Proof. Since P is nontrivial for $\mathcal{L}$, there is some $L_0 \in \mathcal{L}$ so that $P(L_0)$ is false.

Let Σ be large enough, so that $L_0 \subseteq \Sigma^*$, and the problem $L = \Sigma^*$ is undecidable for $L \in \mathcal{L}$.

We show that given any $L \in \mathcal{L}$, with $L \subseteq \Sigma^*$, we can construct a language $L_1 \in \mathcal{L}$, so that $L = \Sigma^*$ iff $P(L_1)$ holds. Thus, the problem $L = \Sigma^*$ for $L \in \mathcal{L}$ reduces to property P for $\mathcal{L}$, and since for Σ big enough, the first problem is undecidable, so is the second.

For any $L \in \mathcal{L}$, with $L \subseteq \Sigma^*$, let

$$L_1 = L_0\#\Sigma^* \cup \Sigma^*\#L.$$

Since $\mathcal{L}$ is effectively closed under union and concatenation with the regular languages, we have $L_1 \in \mathcal{L}$.

If $L = \Sigma^*$, then $L_1 = \Sigma^*\#\Sigma^*$, a regular language, and thus, $P(L_1)$ holds, since P holds for the regular languages.

Conversely, we would like to prove that if $L \neq \Sigma^*$, then $P(L_1)$ is false.

Since $L \neq \Sigma^*$, there is some $w \notin L$. But then,

$$L_1/\#w = L_0.$$

Since P is preserved under quotient by a single letter, by a trivial induction, if $P(L_1)$ holds, then $P(L_0)$ also holds. However, $P(L_0)$ is false, so $P(L_1)$ must be false.

Thus, we proved that $L = \Sigma^*$ iff $P(L_1)$ holds, as claimed. $\square$

Greibach's theorem can be used to show that it is undecidable whether a context-free grammar generates a regular language.

It can also be used to show that it is undecidable whether a context-free language is inherently ambiguous.

9.4 Undecidability of Validity in First-Order Logic

The PCP can also be used to give a quick proof of Church's famous result stating that validity in first-order logic is undecidable. Here we are considering first-order formulae as defined in Section 2.15. Given a first-order language $\mathbf{L}$ consisting of constant symbols c, function symbols f, and predicate symbols P, a *first-order structure* $\mathcal{M}$ consists of a nonempty domain M, of an assigment of some element of $c_{\mathcal{M}} \in M$ to every constant symbol c, of a function $f_{\mathcal{M}}\colon M^n \to M$ to every n-ary function symbol f, and to a boolean-valued function $P_{\mathcal{M}}\colon M^m \to \{\mathbf{T}, \mathbf{F}\}$ to any m-ary predicate symbol P.

Then given any assignment $\rho\colon X \to M$ to the first-order variables $x_i \in X$, we can define recursively the truth value $\varphi_{\mathcal{M}}[\rho]$ of every first-order formula φ. If φ is a sentence, which means that φ has no free variables, then the truth value $\varphi_{\mathcal{M}}[\rho]$ is independent of ρ, so we simply write $\varphi_{\mathcal{M}}$. Details can be found in Gallier [19], Enderton [12], or Shoenfield [45]. The formula φ is *valid in* $\mathcal{M}$ if $\varphi_{\mathcal{M}}[\rho] = \mathbf{T}$ for all ρ. We also say that $\mathcal{M}$ is a *model* of φ and we write

$$\mathcal{M} \models \varphi.$$

The formula φ is *valid* (or *universally valid*) if it is valid in *every* first-order structure $\mathcal{M}$; this denoted by

$$\models \varphi.$$

The *validity problem* in first-order logic is to decide whether there is algorithm to decide whether any first-order formula is valid.

Theorem 9.9. *(Church, 1936) The validity problem for first-order logic is undecidable.*

Proof. The following proof due to R. Floyd is given in Manna [37] (Section 2.16). The proof consists in reducing the PCP over the alphabet $\{0, 1\}$ to the validity problem. Given an instance $S = (U, V)$ of the PCP, we construct a first-order sentence Φ_S (using a computable function) such that S has a solution if and only if Φ_S is valid. Since the PCP is undecidable, so is the validity problem for first-order logic.

For this construction, we need a constant symbol a, two unary function symbols f_0 and f_1, and a binary predicate symbol P. We denote the term

$$f_{s_p}(\cdots(f_{s_2}(f_{s_1}(x))\cdots)$$

as $f_{s_1 s_2 \cdots s_p}$, where $s_i \in \{0,1\}$. Suppose S is the set of pairs

$$S = \{(u_1, v_1), \ldots, (u_m, v_m)\}.$$

The key ingredent is the sentence

$$\begin{aligned}\Phi_S \equiv \Big(\bigwedge_{i=1}^{m} P(f_{u_i}(a), f_{v_i}(a)) \land \forall x \forall y \Big(P(x,y) \Rightarrow \bigwedge_{i=1}^{m} P(f_{u_i}(x), f_{v_i}(x))\Big)\Big)\\ \Rightarrow \exists z P(z,z).\end{aligned}$$

We claim that the PCP S has a solution iff Φ_S is valid.

Step 1. We prove that if Φ_S is valid, then the PCP has a solution. Consider the first-order structure $\mathcal{M}$ with domain $\{0,1\}^*$, with $a_{\mathcal{M}} = \epsilon$, $(f_0)_{\mathcal{M}}$ is concatenation on the right with 0 ($(f_0)_{\mathcal{M}}(x) = x0$), $(f_1)_{\mathcal{M}}$ is concatenation on the right with 1 ($(f_1)_{\mathcal{M}}(x) = x1$), and

$$P(x,y) = \mathbf{T} \quad \text{iff} \quad x = u_{i_1} u_{i_2} \cdots u_{i_n}, \quad y = v_{i_1} v_{i_2} \cdots v_{i_n},$$

for some nonempty sequence $i_1, i_2, \ldots, i_n$ with $1 \le i_j \le m$.

Since Φ_S is valid, it must be valid in $\mathcal{M}$, but then we see immediately that both

$$\bigwedge_{i=1}^{m} P(f_{u_i}(a), f_{v_i}(a))$$

and

$$\forall x \forall y \Big(P(x,y) \Rightarrow \bigwedge_{i=1}^{m} P(f_{u_i}(x), f_{v_i}(x))\Big)$$

are valid in $\mathcal{M}$, thus

$$\exists z P(z,z)$$

is also valid in $\mathcal{M}$. This means that there is some nonempty sequence $i_1, i_2, \ldots, i_n$ with $1 \le i_j \le m$ such that

$$z = u_{i_1} u_{i_2} \cdots u_{i_n} = v_{i_1} v_{i_2} \cdots v_{i_n},$$

and so we have a solution of the PCP.

Step 2. We prove that if the PCP has a solution, then Φ_S is valid. Let $i_1, i_2, \ldots, i_n$ be a nonempty sequence with $1 \le i_j \le m$ such that

$$u_{i_1} u_{i_2} \cdots u_{i_n} = v_{i_1} v_{i_2} \cdots v_{i_n},$$

which means that $i_1, i_2, \ldots, i_n$ is a solution of the PCP S. We prove that for every first-order structure $\mathcal{M}$, if

$$\bigwedge_{i=1}^{m} P(f_{u_i}(a), f_{v_i}(a))$$

and

$$\forall x \forall y \Big(P(x, y) \Rightarrow \bigwedge_{i=1}^{m} P(f_{u_i}(x), f_{v_i}(x))\Big)$$

are valid in $\mathcal{M}$, then

$$\exists z P(z, z)$$

is also valid in $\mathcal{M}$. But then Φ_S is valid in *every* first-order structure $\mathcal{M}$, and thus it is valid.

To finish the proof, assume that $\mathcal{M}$ is any first-order structure such that

$$\bigwedge_{i=1}^{m} P(f_{u_i}(a), f_{v_i}(a)) \tag{$*_1$}$$

and

$$\forall x \forall y \Big(P(x, y) \Rightarrow \bigwedge_{i=1}^{m} P(f_{u_i}(x), f_{v_i}(x))\Big) \tag{$*_2$}$$

are valid in $\mathcal{M}$. Using ($*_1$), by repeated application on ($*_2$), we deduce that

$$P(f_{u_{i_1} u_{i_2} \cdots u_{i_n}}(a), f_{v_{i_1} v_{i_2} \cdots v_{i_n}}(a)),$$

is valid in $\mathcal{M}$. For example, since (u_{i_1}, v_{i_1}) is a pair in the PCP instance, by ($*_1$) the proposition $P(f_{u_{i_1}}(a), f_{v_{i_1}}(a))$ holds, so by ($*_2$) with $x = f_{u_{i_1}}(a)$ and $v = f_{v_{i_1}}(a)$, we get the implication

$$P(f_{u_{i_1}}(a), f_{v_{i_1}}(a)) \Rightarrow \bigwedge_{i=1}^{m} P(f_{u_i}(f_{u_{i_1}}(a)), f_{v_i}(f_{v_{i_1}}(a))),$$

and since $P(f_{u_{i_1}}(a), f_{v_{i_1}}(a))$ holds, we deduce that $\bigwedge_{i=1}^{m} P(f_{u_i}(f_{u_{i_1}}(a)), f_{v_i}(f_{v_{i_1}}(a)))$ holds, and consequently $P(f_{u_{i_2}}(f_{u_{i_1}}(a)), f_{v_{i_2}}(f_{v_{i_1}}(a))) = P(f_{u_{i_1} u_{i_2}}(a), f_{v_{i_1} v_{2_1}}(a))$ holds.

Since by hypothesis

$$u_{i_1} u_{i_2} \cdots u_{i_n} = v_{i_1} v_{i_2} \cdots v_{i_n},$$

we deduce that $\exists z P(z, z)$ is valid in $\mathcal{M}$, and so Φ_S is valid in $\mathcal{M}$, as claimed. □

There are other ways of proving Church's theorem. Among other sources, see Shoenfield [45] (Section 6.8) and Machtey and Young [36] (Chapter 4, theorem 4.3.6). These proofs are rather long and involve complicated arguments. Floyd's proof has the virtue of being quite short and transparent, if we accept the undecidability of the PCP.

Lewis shows the stronger result than even with a *single* unary function symbol f, one constant a, and one binary predicate symbol P, the validity problem is undecidable; see

Lewis [34] (Chapter IIC). Lewis' proof is a very clever reduction of a tiling problem. Lewis' book also contains an extensive classification of undecidable classes of first-order sentences. On the positive side, Dreben and Goldfarb [10] contains a very complete study of classes of first-order sentences for which the validity problem *is* decidable.

Chapter 10

Computational Complexity; $\mathcal{P}$ and $\mathcal{NP}$

10.1 The Class $\mathcal{P}$

In the previous two chapters, we clarified what it means for a problem to be decidable or undecidable. This chapter is heavily inspired by Lewis and Papadimitriou's excellent treatment [35].

In principle, if a problem is decidable, then there is an algorithm (i.e., a procedure that halts for every input) that decides every instance of the problem.

However, from a practical point of view, knowing that a problem is decidable may be useless, if the number of steps (*time complexity*) required by the algorithm is excessive, for example, exponential in the size of the input, or worse.

For instance, consider the *traveling salesman problem*, which can be formulated as follows:

We have a set $\{c_1, \ldots, c_n\}$ of cities, and an $n \times n$ matrix $D = (d_{ij})$ of nonnegative integers, the *distance matrix*, where d_{ij} denotes the distance between c_i and c_j, which means that $d_{ii} = 0$ and $d_{ij} = d_{ji}$ for all $i \neq j$.

The problem is to find a *shortest tour* of the cities, that is, a permutation π of $\{1, \ldots, n\}$ so that the *cost*

$$C(\pi) = d_{\pi(1)\pi(2)} + d_{\pi(2)\pi(3)} + \cdots + d_{\pi(n-1)\pi(n)} + d_{\pi(n)\pi(1)}$$

is as small as possible (minimal).

One way to solve the problem is to consider all possible tours, i.e., $n!$ permutations. Actually, since the starting point is irrelevant, we need only consider $(n-1)!$ tours, but this still grows very fast. For example, when $n = 40$, it turns out that $39!$ exceeds 10^{45}, a huge number.

Consider the 4×4 symmetric matrix given by

$$D = \begin{pmatrix} 0 & 2 & 1 & 1 \\ 2 & 0 & 1 & 1 \\ 1 & 1 & 0 & 3 \\ 1 & 1 & 3 & 0 \end{pmatrix},$$

and the budget $B = 4$. The tour specified by the permutation

$$\pi = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 1 & 4 & 2 & 3 \end{pmatrix}$$

has cost 4, since

$$\begin{aligned} c(\pi) &= d_{\pi(1)\pi(2)} + d_{\pi(2)\pi(3)} + d_{\pi(3)\pi(4)} + d_{\pi(4)\pi(1)} \\ &= d_{14} + d_{42} + d_{23} + d_{31} \\ &= 1 + 1 + 1 + 1 = 4. \end{aligned}$$

The cities in this tour are traversed in the order

$$(1, 4, 2, 3, 1).$$

Remark: The permutation π shown above is described in Cauchy's *two-line notation*,

$$\pi = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 1 & 4 & 2 & 3 \end{pmatrix},$$

where every element in the second row is the image of the element immediately above it in the first row: thus

$$\pi(1) = 1,\ \pi(2) = 4,\ \pi(3) = 2,\ \pi(4) = 3.$$

Thus, to capture the essence of practically feasible algorithms, we must limit our computational devices to run only for a number of steps that is bounded by a *polynomial* in the length of the input.

We are led to the definition of polynomially bounded computational models.

Definition 10.1. A deterministic Turing machine M is said to be *polynomially bounded* if there is a polynomial $p(X)$ so that the following holds: for every input $x \in \Sigma^*$, there is no ID ID_n so that

$$ID_0 \vdash ID_1 \vdash^* ID_{n-1} \vdash ID_n, \quad \text{with} \quad n > p(|x|),$$

where $ID_0 = q_0x$ is the starting ID.

A language $L \subseteq \Sigma^*$ is *polynomially decidable* if there is a polynomially bounded Turing machine that accepts L. The family of all polynomially decidable languages is denoted by $\mathcal{P}$.

Remark: Even though Definition 10.1 is formulated for Turing machines, it can also be formulated for other models, such as RAM programs. The reason is that the conversion of a Turing machine into a RAM program (and vice versa) produces a program (or a machine) whose size is polynomial in the original device.

The following proposition, although trivial, is useful:

Proposition 10.1. *The class $\mathcal{P}$ is closed under complementation.*

Of course, many languages do not belong to $\mathcal{P}$. One way to obtain such languages is to use a diagonal argument. But there are also many natural languages that are not in $\mathcal{P}$, although this may be very hard to prove for some of these languages.

Let us consider a few more problems in order to get a better feeling for the family $\mathcal{P}$.

10.2 Directed Graphs, Paths

Recall that a *directed graph*, G, is a pair $G = (V, E)$, where $E \subseteq V \times V$. Every $u \in V$ is called a *node* (or *vertex*) and a pair $(u, v) \in E$ is called an *edge* of G.

We will restrict ourselves to *simple graphs*, that is, graphs without edges of the form (u, u); equivalently, $G = (V, E)$ is a simple graph if whenever $(u, v) \in E$, then $u \neq v$.

Given any two nodes $u, v \in V$, a *path from u to v* is any sequence of $n+1$ edges ($n \geq 0$)

$$(u, v_1), (v_1, v_2), \ldots, (v_n, v).$$

(If $n = 0$, a path from u to v is simply a single edge, (u, v).)

A graph G is *strongly connected* if for every pair $(u, v) \in V \times V$, there is a path from u to v. A *closed path, or cycle*, is a path from some node u to itself.

We will restrict out attention to finite graphs, i.e. graphs (V, E) where V is a finite set.

Definition 10.2. Given a graph G, an *Eulerian cycle* is a cycle in G that passes through all the nodes (possibly more than once) and every edge of G exactly once. A *Hamiltonian cycle* is a cycle that passes through all the nodes exactly once (note, some edges may not be traversed at all).

Eulerian Cycle Problem: Given a graph G, is there an Eulerian cycle in G?

Hamiltonian Cycle Problem: Given a graph G, is there an Hamiltonian cycle in G?

10.3 Eulerian Cycles

The following graph is a directed graph version of the Königsberg bridge problem, solved by Euler in 1736.

The nodes A, B, C, D correspond to four areas of land in Königsberg and the edges to the seven bridges joining these areas of land.

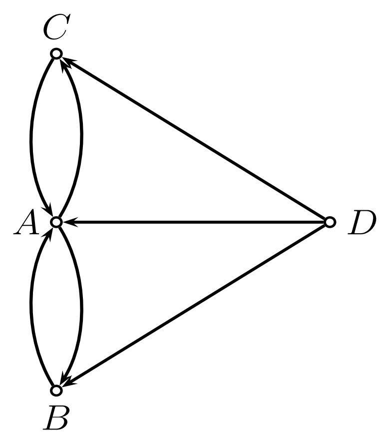


Figure 10.1: A directed graph modeling the Königsberg bridge problem.

The problem is to find a closed path that crosses every bridge exactly once and returns to the starting point.

In fact, the problem is unsolvable, as shown by Euler, because some nodes do not have the same number of incoming and outgoing edges (in the undirected version of the problem, some nodes do not have an even degree.)

It may come as a surprise that the Eulerian Cycle Problem does have a polynomial time algorithm, but that so far, not such algorithm is known for the Hamiltonian Cycle Problem. The reason why the Eulerian Cycle Problem is decidable in polynomial time is the following theorem due to Euler:

Theorem 10.2. *A graph $G = (V, E)$ has an Eulerian cycle iff the following properties hold:*

(1) The graph G is strongly connected.

(2) Every node has the same number of incoming and outgoing edges.

Proving that properties (1) and (2) hold if G has an Eulerian cycle is fairly easy. The converse is harder, but not that bad (try!).

Theorem 10.2 shows that it is necessary to check whether a graph is strongly connected. This can be done by computing the transitive closure of E, which can be done in polynomial time (in fact, $O(n^3)$).

Checking property (2) can clearly be done in polynomial time. Thus, the Eulerian cycle problem is in $\mathcal{P}$. Unfortunately, no theorem analogous to Theorem 10.2 is known for Hamiltonian cycles.

10.4 Hamiltonian Cycles

A game invented by Sir William Hamilton in 1859 uses a regular solid dodecahedron whose twenty vertices are labeled with the names of famous cities.

The player is challenged to "travel around the world" by finding a closed cycle along the edges of the dodecahedron which passes through every city exactly once (this is the undirected version of the Hamiltonian cycle problem).

In graphical terms, assuming an orientation of the edges between cities, the graph D shown in Figure 10.2 is a plane projection of a regular dodecahedron and we want to know if there is a Hamiltonian cycle in this directed graph.

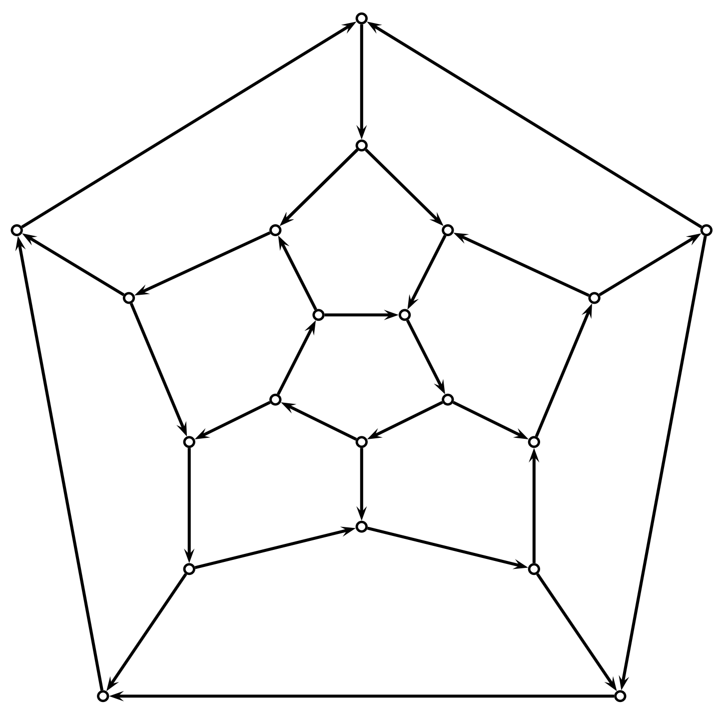


Figure 10.2: A tour "around the world."

Finding a Hamiltonian cycle in this graph does not appear to be so easy!

A solution is shown in Figure 10.3 below.

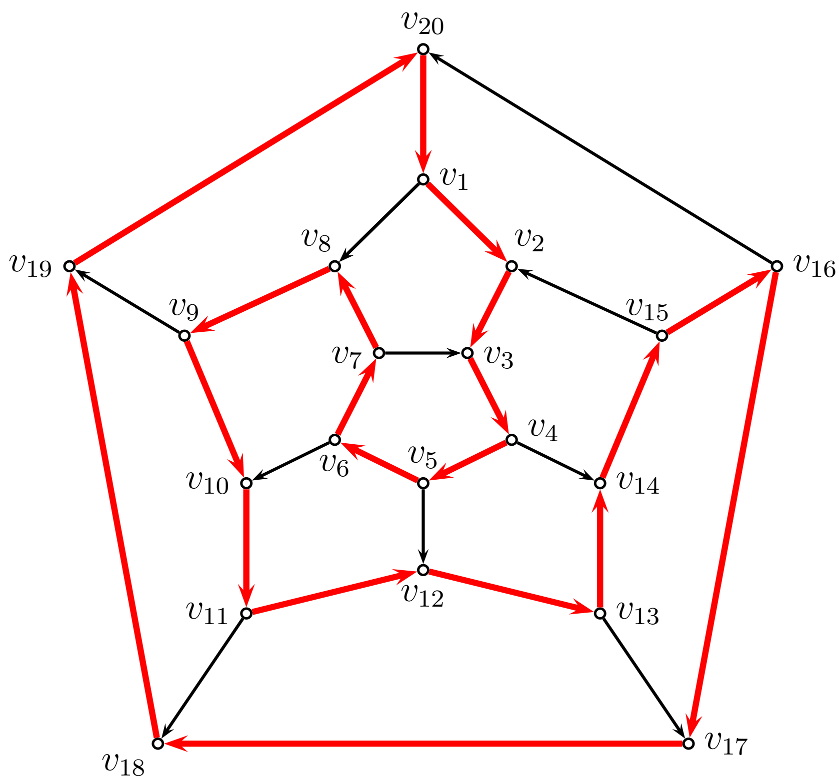


Figure 10.3: A Hamiltonian cycle in D.

Remark: We talked about problems being decidable in polynomial time. Obviously, this is equivalent to deciding some property of a certain class of objects, for example, finite graphs.

Our framework requires that we first encode these classes of objects as strings (or numbers), since $\mathcal{P}$ consists of languages.

Thus, when we say that a property is decidable in polynomial time, we are really talking about the encoding of this property as a language. Thus, we have to be careful about these encodings, but it is rare that encodings cause problems.

10.5 Propositional Logic and Satisfiability

We define the syntax and the semantics of propositions in conjunctive normal form (CNF).

The syntax has to do with the legal form of propositions in CNF. Such propositions are interpreted as truth functions, by assigning truth values to their variables.

We begin by defining propositions in CNF. Such propositions are constructed from a countable set, **PV**, of *propositional (or boolean) variables*, say

$$\mathbf{PV} = \{x_1, x_2, \ldots, \},$$

using the connectives $\wedge$ (and), $\vee$ (or) and $\neg$ (negation).

Definition 10.3. We define a *literal (or atomic proposition)*, L, as $L = x$ or $L = \neg x$, also denoted by $\overline{x}$, where $x \in \mathbf{PV}$.

A *clause*, C, is a disjunction of pairwise distinct literals,

$$C = (L_1 \vee L_2 \vee \cdots \vee L_m).$$

Thus, a clause may also be viewed as a nonempty *set*

$$C = \{L_1, L_2, \ldots, L_m\}.$$

We also have a special clause, the *empty clause*, denoted $\perp$ or $\square$ (or $\{\}$). It corresponds to the truth value false.

A *proposition in CNF, or boolean formula*, P, is a conjunction of pairwise distinct clauses

$$P = C_1 \wedge C_2 \wedge \cdots \wedge C_n.$$

Thus, a boolean formula may also be viewed as a nonempty *set*

$$P = \{C_1, \ldots, C_n\},$$

but this time, the comma is interpreted as conjunction. We also allow the proposition $\perp$, and sometimes the proposition $\top$ (corresponding to the truth value true).

For example, here is a boolean formula:

$$P = \{(x_1 \vee x_2 \vee x_3), (\overline{x_1} \vee x_2), (\overline{x_2} \vee x_3), (\overline{x_3} \vee x_1), (\overline{x_1} \vee \overline{x_2} \vee \overline{x_3})\}.$$

In order to interpret boolean formulae, we use truth assignments.

Definition 10.4. We let $\mathrm{BOOL} = \{\mathbf{F}, \mathbf{T}\}$, the set of *truth values*, where $\mathbf{F}$ stands for false and $\mathbf{T}$ stands for true. A *truth assignment (or valuation)*, v, is any function $v\colon \mathbf{PV} \to \mathrm{BOOL}$.

Example 10.1. The function $v_F\colon \mathbf{PV} \to \mathrm{BOOL}$ given by

$$v_F(x_i) = \mathbf{F} \quad \text{for all } i \geq 1$$

is a truth assignment, and so is the function $v_T\colon \mathbf{PV} \to \text{BOOL}$ given by

$$v_T(x_i) = \mathbf{T} \quad \text{for all } i \geq 1.$$

The function $v\colon \mathbf{PV} \to \text{BOOL}$ given by

$$\begin{aligned} v(x_1) &= \mathbf{T} \\ v(x_2) &= \mathbf{F} \\ v(x_3) &= \mathbf{T} \\ v(x_i) &= \mathbf{T} \quad \text{for all } i \geq 4 \end{aligned}$$

is also a truth assignment.

Definition 10.5. Given a truth assignment $v\colon \mathbf{PV} \to \text{BOOL}$, we define the *truth value* $\widehat{v}(X)$ of a literal, clause, and boolean formula, X, using the following recursive definition:

(1) $\widehat{v}(\perp) = \mathbf{F}$, $\widehat{v}(\top) = \mathbf{T}$.

(2) $\widehat{v}(x) = v(x)$, if $x \in \mathbf{PV}$.

(3) $\widehat{v}(\overline{x}) = \overline{v(x)}$, if $x \in \mathbf{PV}$, where $\overline{v(x)} = \mathbf{F}$ if $v(x) = \mathbf{T}$ and $\overline{v(x)} = \mathbf{T}$ if $v(x) = \mathbf{F}$.

(4) $\widehat{v}(C) = \mathbf{F}$ if C is a clause and iff $\widehat{v}(L_i) = \mathbf{F}$ for all literals L_i in C, otherwise $\mathbf{T}$.

(5) $\widehat{v}(P) = \mathbf{T}$ if P is a boolean formula and iff $\widehat{v}(C_j) = \mathbf{T}$ for all clauses C_j in P, otherwise $\mathbf{F}$.

Since a boolean formula P only contains a finite number of variables, say $\{x_{i_1}, \ldots, x_{i_n}\}$, one should expect that its truth value $\widehat{v}(P)$ depends only on the truth values assigned by the truth assignment v to the variables in the set $\{x_{i_1}, \ldots, x_{i_n}\}$, and this is indeed the case. The following proposition is easily shown by induction on the depth of P (viewed as a tree).

Proposition 10.3. *Let P be a boolean formula containing the set of variables $\{x_{i_1}, \ldots, x_{i_n}\}$. If $v_1\colon \mathbf{PV} \to \text{BOOL}$ and $v_2\colon \mathbf{PV} \to \text{BOOL}$ are any truth assignments agreeing on the set of variables $\{x_{i_1}, \ldots, x_{i_n}\}$, which means that*

$$v_1(x_{i_j}) = v_2(x_{i_j}) \quad \textit{for } j = 1, \ldots, n,$$

then $\widehat{v_1}(P) = \widehat{v_2}(P)$.

In view of Proposition 10.3, given any boolean formula P, we only need to specify the values of a truth assignment v for the variables occurring on P.

Example 10.2. Given the boolean formula

$$P = \{(x_1 \vee x_2 \vee x_3), (\overline{x_1} \vee x_2), (\overline{x_2} \vee x_3), (\overline{x_3} \vee x_1), (\overline{x_1} \vee \overline{x_2} \vee \overline{x_3})\},$$

we only need to specify $v(x_1), v(x_2), v(x_3)$. Thus there are $2^3 = 8$ distinct truth assignments:

$$\begin{array}{ll}
\mathbf{F},\mathbf{F},\mathbf{F} & \mathbf{T},\mathbf{F},\mathbf{F} \\
\mathbf{F},\mathbf{F},\mathbf{T} & \mathbf{T},\mathbf{F},\mathbf{T} \\
\mathbf{F},\mathbf{T},\mathbf{F} & \mathbf{T},\mathbf{T},\mathbf{F} \\
\mathbf{F},\mathbf{T},\mathbf{T} & \mathbf{T},\mathbf{T},\mathbf{T}.
\end{array}$$

In general, there are 2^n distinct truth assignments to n distinct variables.

Example 10.3. Here is an example showing the evaluation of the truth value $\widehat{v}(P)$ for the boolean formula

$$\begin{aligned}
P &= (x_1 \lor x_2 \lor x_3) \land (\overline{x_1} \lor x_2) \land (\overline{x_2} \lor x_3) \land (\overline{x_3} \lor x_1) \land (\overline{x_1} \lor \overline{x_2} \lor \overline{x_3}) \\
&= \{(x_1 \lor x_2 \lor x_3), (\overline{x_1} \lor x_2), (\overline{x_2} \lor x_3), (\overline{x_3} \lor x_1), (\overline{x_1} \lor \overline{x_2} \lor \overline{x_3})\},
\end{aligned}$$

and the truth assignment

$$v(x_1) = \mathbf{T}, \quad v(x_2) = \mathbf{F}, \quad v(x_3) = \mathbf{F}.$$

For the literals, we have

$$\widehat{v}(x_1) = \mathbf{T}, \quad \widehat{v}(x_2) = \mathbf{F}, \quad \widehat{v}(x_3) = \mathbf{F}, \quad \widehat{v}(\overline{x_1}) = \mathbf{F}, \quad \widehat{v}(\overline{x_2}) = \mathbf{T}, \quad \widehat{v}(\overline{x_3}) = \mathbf{T},$$

for the clauses

$$\begin{aligned}
\widehat{v}(x_1 \lor x_2 \lor x_3) &= \widehat{v}(x_1) \lor \widehat{v}(x_2) \lor \widehat{v}(x_3) = \mathbf{T} \lor \mathbf{F} \lor \mathbf{F} = \mathbf{T}, \\
\widehat{v}(\overline{x_1} \lor x_2) &= \widehat{v}(\overline{x_1}) \lor \widehat{v}(x_2) = \mathbf{F} \lor \mathbf{F} = \mathbf{F}, \\
\widehat{v}(\overline{x_2} \lor x_3) &= \widehat{v}(\overline{x_2}) \lor \widehat{v}(x_3) = \mathbf{T} \lor \mathbf{F} = \mathbf{T}, \\
\widehat{v}(\overline{x_3} \lor x_1) &= \widehat{v}(\overline{x_3}) \lor \widehat{v}(x_1) = \mathbf{T} \lor \mathbf{T} = \mathbf{T}, \\
\widehat{v}(\overline{x_1} \lor \overline{x_2} \lor \overline{x_3}) &= \widehat{v}(\overline{x_1}) \lor \widehat{v}(\overline{x_2}) \lor \widehat{v}(\overline{x_3}) = \mathbf{F} \lor \mathbf{T} \lor \mathbf{T} = \mathbf{T},
\end{aligned}$$

and for the conjunction of the clauses,

$$\begin{aligned}
\widehat{v}(P) &= \widehat{v}(x_1 \lor x_2 \lor x_3) \land \widehat{v}(\overline{x_1} \lor x_2) \land \widehat{v}(\overline{x_2} \lor x_3) \land \widehat{v}(\overline{x_3} \lor x_1) \land \widehat{v}(\overline{x_1} \lor \overline{x_2} \lor \overline{x_3}) \\
&= \mathbf{T} \land \mathbf{F} \land \mathbf{T} \land \mathbf{T} \land \mathbf{T} = \mathbf{F}.
\end{aligned}$$

Therefore, $\widehat{v}(P) = \mathbf{F}$.

Definition 10.6. We say that a truth assignment v *satisfies* a boolean formula P, if $\widehat{v}(P) = \mathbf{T}$. In this case, we also write

$$v \models P.$$

A boolean formula P is *satisfiable* if $v \models P$ for some truth assignment v, otherwise, it is *unsatisfiable*. A boolean formula P is *valid (or a tautology)* if $v \models P$ for all truth assignments v, in which case we write

$$\models P.$$

One should check that the boolean formula

$$P = \{(x_1 \vee x_2 \vee x_3), (\overline{x_1} \vee x_2), (\overline{x_2} \vee x_3), (\overline{x_3} \vee x_1), (\overline{x_1} \vee \overline{x_2} \vee \overline{x_3})\}$$

is **un**satisfiable.

One may think that it is easy to test whether a proposition is satisfiable or not. Try it, it is not that easy!

As a matter of fact, the *satisfiability problem*, testing whether a boolean formula is satisfiable, also denoted SAT, is not known to be in $\mathcal{P}$. Moreover, it is an $\mathcal{NP}$-complete problem (see Section 10.6). Most people believe that the satisfiability problem is **not** in $\mathcal{P}$, but a proof still eludes us!

Before we explain what is the class $\mathcal{NP}$, we state the following result.

Proposition 10.4. *The satisfiability problem for clauses containing at most two literals (2-satisfiability, or* 2-SAT*) is solvable in polynomial time.*

Proof sketch. The first step consists in observing that if every clause in P contains at most two literals, then we can reduce the problem to testing satisfiability when every clause has exactly two literals.

Indeed, if P contains some clause (x), then any valuation satisfying P must make x true. Then all clauses containing x will be true, and we can delete them, whereas we can delete $\overline{x}$ from every clause containing it, since $\overline{x}$ is false.

Similarly, if P contains some clause $(\overline{x})$, then any valuation satisfying P must make x false. Then all clauses containing $\overline{x}$ will be true, and we can delete them, whereas we can delete x from every clause containing it.

Thus in a finite number of steps, either all the clauses were satisfied and P is satisfiable, or we get the empty clause and P is unsatisfiable, or we get a set of clauses with exactly two literals. The number of steps is clearly linear in the number of literals in P. Here are some examples illustrating the three possible oucomes.

(1) Consider the conjunction of clauses

$$P_1 = (x_1 \vee \overline{x_2}) \wedge (x_2) \wedge (x_1 \vee x_3) \wedge (x_2 \vee \overline{x_3}).$$

We must set x_2 to $\mathbf{T}$, so $(x_1 \vee \overline{x_2})$ becomes (x_1) and $(x_2 \vee \overline{x_3})$ becomes $\mathbf{T}$ and can be deleted. We now have

$$P = (x_1) \wedge (x_1 \vee x_3) \wedge (\overline{x_3}).$$

We must set x_1 to $\mathbf{T}$, so $(x_1 \vee x_3)$ becomes $\mathbf{T}$ and can be deleted. We must also set x_3 to $\mathbf{F}$, so $(\overline{x_3})$ becomes $\mathbf{T}$ and all the clauses are satisfied.

(2) Consider the conjunction of clauses

$$P_2 = (x_1) \wedge (x_3) \wedge (\overline{x_1} \vee x_2) \wedge (\overline{x_2} \vee \overline{x_3}).$$

We must set x_1 to **T**, so $(\overline{x_1} \vee x_2)$ becomes (x_2). We now have

$$(x_3) \wedge (x_2) \wedge (\overline{x_2} \vee \overline{x_3}).$$

We must set x_3 to **T**, so $(\overline{x_2} \vee \overline{x_3})$ becomes $(\neg x_2)$. We now have

$$(x_2) \wedge (\overline{x_2}).$$

We must set x_2 to **T**, so $(\overline{x_2})$ becomes the empty clause, which means that P_2 is unsatisfiable.

For the second step, we construct a directed graph from P. The purpose of this graph is to propagate truth. The nodes of this graph are the literals in P, and edges are defined as follows:

(1) For every clause $(\overline{x} \vee y)$, there is an edge from x to y and an edge from $\overline{y}$ to $\overline{x}$.

(2) For every clause $(x \vee y)$, there is an edge from $\overline{x}$ to y and an edge from $\overline{y}$ to x

(3) For every clause $(\overline{x} \vee \overline{y})$, there is an edge from x to $\overline{y}$ and an edge from y to $\overline{x}$.

Then it can be shown that P is unsatisfiable iff there is some x so that there is a cycle containing x and $\overline{x}$. As a consequence, 2-satisfiability is in $\mathcal{P}$. $\square$

Example 10.4. Consider the following conjunction of clauses:

$$P = (x_1 \vee \overline{x_2}) \wedge (x_1 \vee x_2) \wedge (x_2 \vee \overline{x_3}).$$

It is satisfied by any valuation v such that $v(x_1) = \mathbf{T}$, and if $v(x_2) = \mathbf{F}$ then $v(x_3) = \mathbf{F}$. The construction of the graph associated with P is shown in Figure 10.4.

10.6 The Class $\mathcal{NP}$, Polynomial Reducibility, $\mathcal{NP}$-Completeness

One will observe that the hard part in trying to solve either the Hamiltonian cycle problem or the satisfiability problem, SAT, is to *find* a solution, but that *checking* that a candidate solution is indeed a solution can be done easily in polynomial time.

This is the essence of problems that can be solved *nondetermistically* in polynomial time: a solution can be guessed and then checked in polynomial time.

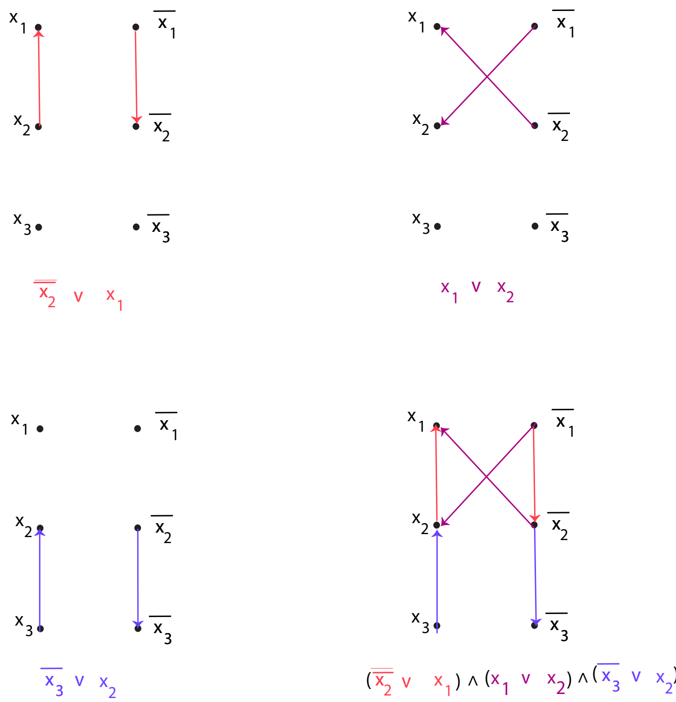


Figure 10.4: The graph coresponding to the clauses of Example 10.4.

Definition 10.7. A nondeterministic Turing machine M is said to be *polynomially bounded* if there is a polynomial $p(X)$ so that the following holds: For every input $x \in \Sigma^*$, there is no ID ID_n so that

$$ID_0 \vdash ID_1 \vdash^* ID_{n-1} \vdash ID_n, \quad \text{with} \quad n > p(|x|),$$

where $ID_0 = q_0x$ is the starting ID.

A language $L \subseteq \Sigma^*$ is *nondeterministic polynomially decidable* if there is a polynomially bounded nondeterministic Turing machine that accepts L. The family of all nondeterministic polynomially decidable languages is denoted by $\mathcal{NP}$.

Of course, we have the inclusion

$$\mathcal{P} \subseteq \mathcal{NP},$$

but whether or not we have equality is one of the most famous open problems of theoretical computer science and mathematics.

In fact, the question $\mathcal{P} \neq \mathcal{NP}$ is one of the open problems listed by the CLAY Institute, together with the Poincaré conjecture and the Riemann hypothesis, among other problems, and for which *one million dollar* is offered as a reward! Actually the Poincaré conjecture was setlled by G. Perelman in 2006, but he rejected receiving the prize in 2010! He also declined the Fields Medal which was awarded to him in 2006.

It is easy to check that SAT is in $\mathcal{NP}$, and so is the Hamiltonian cycle problem.

As we saw in recursion theory, where we introduced the notion of many-one reducibility, in order to compare the "degree of difficulty" of problems, it is useful to introduce the notion of reducibility and the notion of a complete set.

Definition 10.8. A function $f\colon \Sigma^* \to \Sigma^*$ is *polynomial-time computable* if there is a polynomial $p(X)$ so that the following holds: there is a deterministic Turing machine M computing it so that for every input $x \in \Sigma^*$, there is no ID ID_n so that

$$ID_0 \vdash ID_1 \vdash^* ID_{n-1} \vdash ID_n, \quad \text{with} \quad n > p(|x|),$$

where $ID_0 = q_0 x$ is the starting ID.

Given two languages $L_1, L_2 \subseteq \Sigma^*$, *a polynomial-time reduction from L_1 to L_2* is a polynomial-time computable function $f\colon \Sigma^* \to \Sigma^*$ so that for all $u \in \Sigma^*$,

$$u \in L_1 \quad \text{iff} \quad f(u) \in L_2.$$

The notation $L_1 \leq_P L_2$ is often used to denote the fact that there is polynomial-time reduction from L_1 to L_2. Sometimes, the notation $L_1 \leq_m^P L_2$ is used to stress that this is a many-to-one reduction (that is, f is not necessarily injective). This type of reduction is also known as a *Karp reduction*.

A polynomial reduction $f\colon \Sigma^* \to \Sigma^*$ from a language L_1 to a language L_2 is a method that converts in polynomial time every string $u \in \Sigma^*$ (viewed as an instance of a problem A encoded by language L_1) to a string $f(u) \in \Sigma^*$ (viewed as an instance of a problem B encoded by language L_2) in such way that membership in L_1, that is $u \in L_1$, is equivalent to membership in L_2, that is $f(u) \in L_2$.

As a consequence, if we have a procedure to decide membership in L_2 (to solve every instance of problem B), then we have a procedure for solving membership in L_1 (to solve every instance of problem A), since given any $u \in L_1$, we can first apply f to u to produce $f(u)$, and then apply our procedure to decide whether $f(u) \in L_2$; the defining property of f says that this is equivalent to deciding whether $u \in L_1$. Furthermore, if the procedure for deciding membership in L_2 runs deterministically in polynomial time, since f runs deterministically in polynomial time, so does the procedure for deciding membership in L_1, and similarly if the procedure for deciding membership in L_2 runs non deterministically in polynomial time.

For the above reason, we see that membership in L_2 can be considered at least as hard as membership in L_1, since any method for deciding membership in L_2 yields a method for deciding membership in L_1. Thus, if we view L_1 an encoding a problem A and L_2 as encoding a problem B, then B is at least as hard as A.

The following version of Proposition 5.16 for polynomial-time reducibility is easy to prove.

Proposition 10.5. *Let A, B, C be subsets of $\mathbb{N}$ (or Σ^*). The following properties hold:*

(1) If $A \leq_P B$ and $B \leq_P C$, then $A \leq_P C$.

(2) If $A \leq_P B$ then $\overline{A} \leq_P \overline{B}$.

(3) If $A \leq_P B$ and $B \in \mathcal{NP}$, then $A \in \mathcal{NP}$.

(4) If $A \leq_P B$ and $A \notin \mathcal{NP}$, then $B \notin \mathcal{NP}$.

(5) If $A \leq_P B$ and $B \in \mathcal{P}$, then $A \in \mathcal{P}$.

(6) If $A \leq_P B$ and $A \notin \mathcal{P}$, then $B \notin \mathcal{P}$.

Intuitively, we see that if L_1 is a hard problem and L_1 can be reduced to L_2 in polynomial time, then L_2 is also a hard problem.

For example, one can construct a polynomial reduction *from the Hamiltonian cycle problem to the satisfiability problem* SAT. Given a directed graph $G = (V, E)$ with n nodes, say $V = \{1, \ldots, n\}$, we need to construct in polynomial time a set $F = \tau(G)$ of clauses such that G has a Hamiltonian cycle iff $\tau(G)$ is satisfiable. We need to describe a permutation of the nodes that forms a Hamiltonian cycle. For this we introduce n^2 boolean variables x_{ij}, with the intended interpretation that x_{ij} *is true iff node i is the jth node in a Hamiltonian cycle.*

To express that at least one node must appear as the jth node in a Hamiltonian cycle, we have the n clauses

$$(x_{1j} \vee x_{2j} \vee \cdots \vee x_{nj}), \quad 1 \leq j \leq n. \tag{1}$$

The conjunction of these clauses is satisfied iff for every $j = 1, \ldots, n$ there is some node i which is the jth node in the cycle. These n clauses can be produced in time $O(n^2)$.

To express that only one node appears in the cycle, we have the clauses

$$(\overline{x_{ij}} \vee \overline{x_{kj}}), \quad 1 \leq i, j, k \leq n,\ i \neq k. \tag{2}$$

Since $(\overline{x_{ij}} \vee \overline{x_{kj}})$ is equivalent to $\overline{(x_{ij} \wedge x_{kj})}$, each such clause asserts that no two distinct nodes may appear as the jth node in the cycle. Let S_1 be the set of all clauses of type (1) or (2). These n^3 clauses can be produced in time $O(n^3)$.

The conjunction of the clauses in S_1 assert that exactly one node appear at the jth node in the Hamiltonian cycle. We still need to assert that each node i appears exactly once in the cycle. For this, we have the clauses

$$(x_{i1} \vee x_{i2} \vee \cdots \vee x_{in}), \quad 1 \leq i \leq n, \tag{3}$$

and

$$(\overline{x_{ij}} \vee \overline{x_{ik}}), \quad 1 \leq i, j, k \leq n,\ j \neq k. \tag{4}$$

Let S_2 be the set of all clauses of type (3) or (4). These n^3 clauses can be produced in time $O(n^3)$.

The conjunction of the clauses in $S_1 \cup S_2$ asserts that the x_{ij} represents a bijection of $\{1, 2, \ldots, n\}$, in the sense that for any truth assigment v satisfying all these clauses, $i \mapsto j$ iff $v(x_{ij}) = \mathbf{T}$ defines a bijection of $\{1, 2, \ldots, n\}$.

It remains to assert that this permutation of the nodes is a Hamiltonian cycle, which means that if x_{ij} and x_{kj+1} are both true then there there must be an edge (i, k). By contrapositive, this equivalent to saying that if (i, k) is *not* an edge of G, then $\overline{(x_{ij} \wedge x_{kj+1})}$ is true, which as a clause is equivalent to $(\overline{x_{ij}} \vee \overline{x_{kj+1}})$.

Therefore, for all (i, k) such that $(i, k) \notin E$ (with $i, k \in \{1, 2, \ldots, n\}$), we have the clauses

$$(\overline{x_{ij}} \vee \overline{x_{k\, j+1\ (\mathrm{mod}\ n)}}), \quad j = 1, \ldots, n. \tag{5}$$

Let S_3 be the set of clauses of type (5). These n clauses can be produced in time $O(n^2)$.

The conjunction of all the clauses in $S_1 \cup S_2 \cup S_3$ is the boolean formula $F = \tau(G)$. It can be produced in time $O(n^3)$.

We leave it as an exercise to prove that G has a Hamiltonian cycle iff $F = \tau(G)$ is satisfiable.

Example 10.5. Here is an example of a graph with four nodes and four edges shown in Figure 10.5. The Hamiltonian circuit is (x_4, x_3, x_1, x_2).

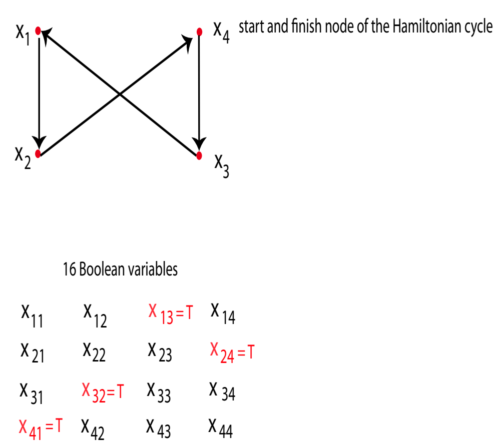


Figure 10.5: A directed graph with a Hamiltonian

It is also possible to construct a reduction of the satisfiability problem to the Hamiltonian cycle problem but this is harder. It is easier to construct this reduction in two steps by introducing an intermediate problem, the exact cover problem, and to provide a polynomial reduction from the satisfiability problem to the exact cover problem, and a polynomial reduction from the exact cover problem to the Hamiltonian cycle problem. These reductions are carried out in Section 11.2.

The above construction of a set $F = \tau(G)$ of clauses from a graph G asserting that G has a Hamiltonian cycle iff F is satisfiable illustrates the expressive power of propositional logic.

Remarkably, *every* language in $\mathcal{NP}$ can be reduced to SAT. Thus, SAT is a hardest problem in $\mathcal{NP}$ (since it is in $\mathcal{NP}$).

Definition 10.9. A language L is *$\mathcal{NP}$-hard* if there is a polynomial reduction from every language $L_1 \in \mathcal{NP}$ to L. A language L is *$\mathcal{NP}$-complete* if $L \in \mathcal{NP}$ and L is $\mathcal{NP}$-hard.

Thus, an $\mathcal{NP}$-hard language is as hard to decide as any language in $\mathcal{NP}$.

Remark: There are $\mathcal{NP}$-hard languages that do not belong to $\mathcal{NP}$. Such problems are really difficult. Two standard examples are K_0 and K, which encode the halting problem. Since K_0 and K are not computable, they can't be in $\mathcal{NP}$. Furthermore, since every language L in $\mathcal{NP}$ is accepted nondeterminsticaly in polynomial time $p(X)$, for some polynomial $p(X)$, for every input w we can try all computations of length at most $p(|w|)$ (there can be exponentially many, but only a finite number), so every language in $\mathcal{NP}$ is computable. Finally, it is shown in Theorem 5.17 that K_0 and K are complete with respect to many-one reducibility, so in particular they are $\mathcal{NP}$-hard. An example of a computable $\mathcal{NP}$-hard language not in $\mathcal{NP}$ will be described after Theorem 10.7.

The importance of $\mathcal{NP}$-complete problems stems from the following theorem which follows immediately from Proposition 10.5.

Theorem 10.6. *Let L be an $\mathcal{NP}$-complete language. Then $\mathcal{P} = \mathcal{NP}$ iff $L \in \mathcal{P}$.*

There are analogies between $\mathcal{P}$ and the class of computable sets, and $\mathcal{NP}$ and the class of listable sets, but there are also important differences. One major difference is that the family of computable sets is properly contained in the family of listable sets, but it is an open problem whether $\mathcal{P}$ is properly contained in $\mathcal{NP}$. We also know that a set L is computable iff both L and $\overline{L}$ are listable, but it is also an open problem whether if both $L \in \mathcal{NP}$ and $\overline{L} \in \mathcal{NP}$, then $L \in \mathcal{P}$. This suggests defining

$$\mathrm{co}\mathcal{NP} = \{\overline{L} \mid L \in \mathcal{NP}\},$$

that is, co$\mathcal{NP}$ consists of all complements of languages in $\mathcal{NP}$. Since $\mathcal{P} \subseteq \mathcal{NP}$ and $\mathcal{P}$ is closed under complementation,

$$\mathcal{P} \subseteq \mathrm{co}\mathcal{NP},$$

and thus

$$\mathcal{P} \subseteq \mathcal{NP} \cap \mathrm{co}\mathcal{NP},$$

but nobody knows whether the inclusion is proper. There are problems in $\mathcal{NP} \cap \text{co}\mathcal{NP}$ not known to be in $\mathcal{P}$; see Section 11.3. It is unknown whether $\mathcal{NP}$ is closed under complementation, that is, nobody knows whether $\mathcal{NP} = \text{co}\mathcal{NP}$. This is considered unlikely. We will come back to co$\mathcal{NP}$ in Section 11.3.

Next we prove a famous theorem of Steve Cook and Leonid Levin (proven independently): SAT is $\mathcal{NP}$-complete.

10.7 The Bounded Tiling Problem is $\mathcal{NP}$-Complete

Instead of showing directly that SAT is $\mathcal{NP}$-complete, which is rather complicated, we proceed in two steps, as suggested by Lewis and Papadimitriou.

(1) First, we define a tiling problem adapted from H. Wang (1961) by Harry Lewis, and we prove that it is $\mathcal{NP}$-complete.

(2) We show that the tiling problem can be reduced to SAT.

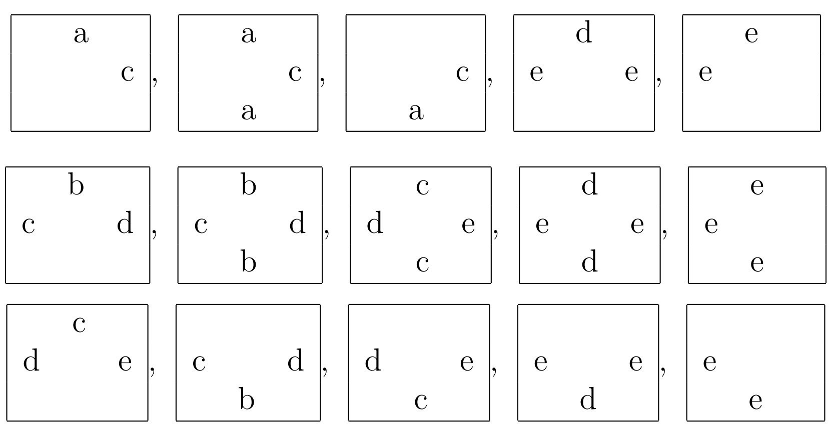
We are given a finite set $\mathcal{T} = \{t_1, \ldots, t_p\}$ of *tile patterns*, for short, *tiles*. We assume that these tiles are unit squares. Copies of these tile patterns may be used to tile a rectangle of predetermined size $2s \times s$ ($s > 1$). However, there are constraints on the way that these tiles may be adjacent horizontally and vertically.

The *horizontal constraints* are given by a relation $H \subseteq \mathcal{T} \times \mathcal{T}$, and the *vertical constraints* are given by a relation $V \subseteq \mathcal{T} \times \mathcal{T}$.

Thus, a *tiling system* is a triple $T = (\mathcal{T}, V, H)$ with V and H as above.

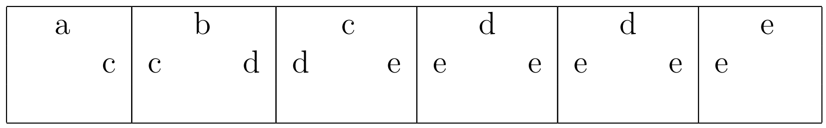
The bottom row of the rectangle of tiles is specified before the tiling process begins.

Example 10.6. For example, consider the following tile patterns:

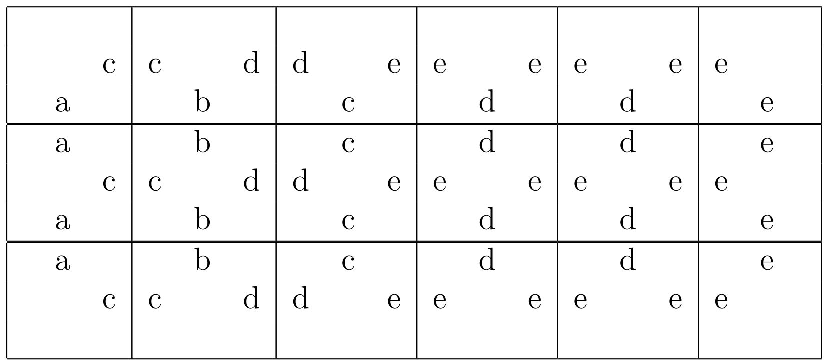


The horizontal and the vertical constraints are that the letters on adjacent edges match (blank edges do not match).

For $s = 3$, given the bottom row



we have the tiling shown below:



Formally, the problem is then as follows:

The Bounded Tiling Problem

Given any tiling system $(\mathcal{T}, V, H)$, any integer $s > 1$, and any initial row of tiles σ_0 (of length $2s$)

$$\sigma_0\colon \{1, 2, \ldots, s, s+1, \ldots, 2s\} \to \mathcal{T},$$

find a $2s \times s$-tiling σ extending σ_0, i.e., a function

$$\sigma\colon \{1, 2, \ldots, s, s+1, \ldots, 2s\} \times \{1, \ldots, s\} \to \mathcal{T}$$

so that

(1) $\sigma(m, 1) = \sigma_0(m)$, for all m with $1 \leq m \leq 2s$.

(2) $(\sigma(m, n), \sigma(m+1, n)) \in H$, for all m with $1 \leq m \leq 2s - 1$, and all n, with $1 \leq n \leq s$.

(3) $(\sigma(m, n), \sigma(m, n+1)) \in V$, for all m with $1 \leq m \leq 2s$, and all n, with $1 \leq n \leq s - 1$.

Formally, an *instance of the tiling problem* is a triple $((\mathcal{T}, V, H), \widehat{s}, \sigma_0)$, where $(\mathcal{T}, V, H)$ is a tiling system, $\widehat{s}$ is the string representation of the number $s \geq 2$, in binary and σ_0 is an initial row of tiles (the bottom row).

For example, if $s = 1025$ (as a decimal number), then its binary representation is $\widehat{s} = 10000000001$. The length of $\widehat{s}$ is $\log_2 s + 1$.

Recall that the input must be a string. This is why the number s is represented by a string in binary. If we only included a *single* tile σ_0 in position $(s+1, 1)$, then the length of the input $((\mathcal{T}, V, H), \widehat{s}, \sigma_0)$ would be $\log_2 s + 1 + C + 1 = \log_2 s + C + 2$ for some constant C corresponding to the length of the string encoding $(\mathcal{T}, V, H)$.

However, the rectangular grid has size $2s^2$, which is *exponential* in the length $\log_2 s + C + 2$ of the input $((\mathcal{T}, V, H), \widehat{s}, \sigma_0)$. Thus, it is impossible to check in polynomial time that a proposed solution is a tiling.

However, if we include in the input the bottom row σ_0 of length $2s$, then the length of input is $\log_2 s + 1 + C + 2s = \log_2 s + C + 2s + 1$ and the size $2s^2$ or the grid is indeed polynomial in the size of the input.

Theorem 10.7. *The tiling problem defined earlier is $\mathcal{NP}$-complete.*

Proof. Let $L \subseteq \Sigma^*$ be any language in $\mathcal{NP}$ and let u be any string in Σ^*. Assume that L is accepted in polynomial time bounded by $p(|u|)$.

We show how to construct an instance of the tiling problem, $((\mathcal{T}, V, H)_L, \widehat{s}, \sigma_0)$, where $s = p(|u|) + 2$, and where the bottom row encodes the starting ID, so that $u \in L$ iff the tiling problem $((\mathcal{T}, V, H)_L, \widehat{s}, \sigma_0)$ has a solution.

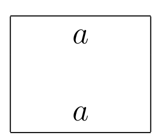
First, note that the problem is indeed in $\mathcal{NP}$, since we have to guess a rectangle of size $2s^2$, and that checking that a tiling is legal can indeed be done in $O(s^2)$, where s is *bounded by the the size of the input* $((\mathcal{T}, V, H), \widehat{s}, \sigma_0)$, since the input contains the bottom row of $2s$ symbols (this is the reason for including the bottom row of $2s$ tiles in the input!).

The idea behind the definition of the tiles is that, in a solution of the tiling problem, the labels on the horizontal edges between two adjacent rows represent a legal ID, *upav*. In a given row, the labels on vertical edges of adjacent tiles keep track of the change of state and direction.

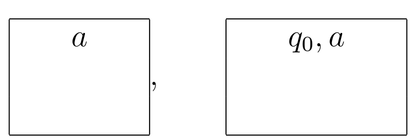
Let Γ be the tape alphabet of the TM, M. As before, we assume that M signals that it accepts u by halting with the output 1 (true).

From M, we create the following tiles:

(1) For every $a \in \Gamma$, tiles

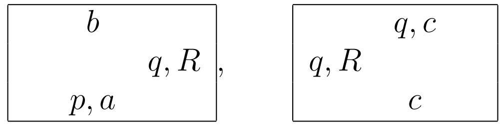


(2) For every $a \in \Gamma$, the bottom row uses tiles

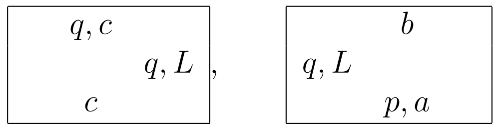


where q_0 is the start state.

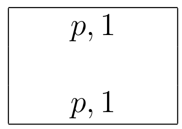
(3) For every instruction $(p, a, b, R, q) \in \delta$, for every $c \in \Gamma$, tiles



(4) For every instruction $(p, a, b, L, q) \in \delta$, for every $c \in \Gamma$, tiles



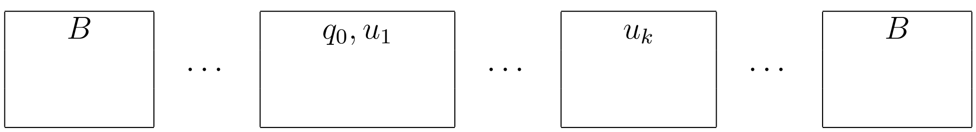
(5) For every halting state, p, tiles



The purpose of tiles of type (5) is to fill the $2s \times s$ rectangle iff M accepts u. Since $s = p(|u|) + 2$ and the machine runs for at most $p(|u|)$ steps, the $2s \times s$ rectangle can be tiled iff $u \in L$.

The vertical and the horizontal constraints are that adjacent edges have the same label (or no label).

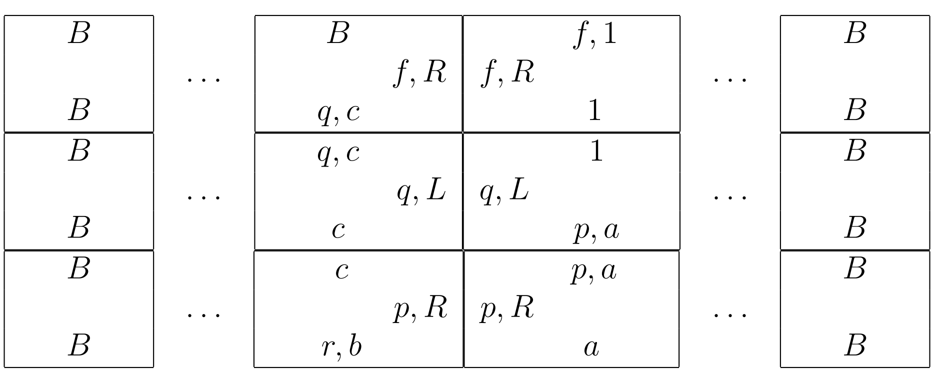
If $u = u_1 \cdots u_k$, the initial bottom row σ_0, of length $2s$, is



where the tile labeled q_0, u_1 is in position $s + 1$.

The example below illustrates the construction:

Example 10.7.



We claim that $u = u_1 \cdots u_k$ is accepted by M iff the tiling problem just constructed has a solution.

The upper horizontal edge of the first (bottom) row of tiles represents the starting configuation $B^s q_0 u B^{s-|u|}$. By induction, we see that after i ($i \leq p(|u|) = s - 2$) steps the upper horizontal edge of the $(i+1)$th row of tiles represents the current ID $upav$ reached by the Turing machine; see Example 10.7. Since the machine runs for at most $p(|u|)$ steps and since $s = p(|u|) + 2$, when the computation stops, at most the lowest $p(|u|) + 1 = s - 1$ rows of the the $2s \times s$ rectangle have been tiled. Assume the machine M stops after $r \leq s - 2$ steps. Then the lowest $r + 1$ rows have been tiled, and since no further instruction can be executed (since the machine entered a halting state), the remaining $s - r - 1$ rows can be filled iff tiles of type (5) can be used iff the machine stopped in an ID containing a pair $p\,1$ where p is a halting state. Therefore, the machine M accepts u iff the $2s \times s$ rectangle can be tiled. $\square$

Remark:

(1) The problem becomes harder if we only specify a *single* tile σ_0 as input, instead of a row of length $2s$. If s is specified in binary (or any other base, but not in tally notation), then the $2s^2$ grid has size exponential in the length $\log_2 s + C + 2$ of the input $((\mathcal{T}, V, H), \widehat{s}, \sigma_0)$, and this tiling problem is actually $\mathcal{NEXP}$-complete! The class $\mathcal{NEXP}$ is the family of languages that can be accepted by a nondeterministic Turing machine that runs in time bounded by $2^{p(|x|)}$, for every x, where p is a polynomial; see the remark after Definition 11.5. By the time hierarchy theorem (Cook, Seiferas, Fischer, Meyer, Zak), it is known that $\mathcal{NP}$ is properly contained in $\mathcal{NEXP}$; see Papadimitriou [38] (Chapters 7 and 20) and Arora and Barak [3] (Chapter 3, Section 3.2). Then the tiling problem with a single tile as input is a computable $\mathcal{NP}$-hard problem not in $\mathcal{NP}$.

(2) If we relax the finiteness condition and require that the entire upper half-plane be tiled, i.e., for every $s > 1$, there is a solution to the $2s \times s$-tiling problem, then the problem is undecidable.

In 1972, Richard Karp published a list of twenty one $\mathcal{NP}$-complete problems.

10.8 The Cook–Levin Theorem: SAT is $\mathcal{NP}$-Complete

We finally prove the Cook-Levin theorem.

Theorem 10.8. *(Cook, 1971, Levin, 1973) The satisfiability problem* SAT *is $\mathcal{NP}$-complete.*

Proof. We reduce the tiling problem to SAT. Given a tiling problem, $((\mathcal{T}, V, H), \widehat{s}, \sigma_0)$, we introduce boolean variables

$$x_{mnt},$$

for all m with $1 \leq m \leq 2s$, all n with $1 \leq n \leq s$, and all tiles $t \in \mathcal{T}$.

The intuition is that $x_{mnt} = \mathbf{T}$ iff tile t occurs in some tiling σ so that $\sigma(m,n) = t$.

We define the following clauses:

(1) For all m, n in the correct range, as above,

$$(x_{mnt_1} \vee x_{mnt_2} \vee \cdots \vee x_{mnt_p}),$$

for all p tiles in $\mathcal{T}$.

This clause states that every position in σ is tiled.

(2) For any two distinct tiles $t \neq t' \in \mathcal{T}$, for all m, n in the correct range, as above,

$$(\overline{x}_{mnt} \vee \overline{x}_{mnt'}).$$

This clause states that a position may not be occupied by more than one tile.

(3) For every pair of tiles $(t, t') \in \mathcal{T} \times \mathcal{T} - H$, for all m with $1 \leq m \leq 2s - 1$, and all n, with $1 \leq n \leq s$,

$$(\overline{x}_{mnt} \vee \overline{x}_{m+1\,nt'}).$$

This clause enforces the horizontal adjacency constraints.

(4) For every pair of tiles $(t, t') \in \mathcal{T} \times \mathcal{T} - V$, for all m with $1 \leq m \leq 2s$, and all n, with $1 \leq n \leq s - 1$,

$$(\overline{x}_{mnt} \vee \overline{x}_{m\,n+1\,t'}).$$

This clause enforces the vertical adjacency constraints.

(5) For all m with $1 \leq m \leq 2s$,

$$(x_{m1\sigma_0(m)}).$$

This clause states that the bottom row is correctly tiled with σ_0.

It is easily checked that the tiling problem has a solution iff the conjunction of the clauses just defined is satisfiable. Thus, SAT is $\mathcal{NP}$-complete. $\square$

We sharpen Theorem 10.8 to prove that 3-SAT is also $\mathcal{NP}$-complete. This is the satisfiability problem for clauses containing at most three literals.

We know that we can't go further and retain $\mathcal{NP}$-completeteness, since 2-SAT is in $\mathcal{P}$.

Theorem 10.9. *(Cook, 1971) The satisfiability problem* 3-SAT *is $\mathcal{NP}$-complete.*

Proof. We have to break "long clauses"

$$C = (L_1 \vee \cdots \vee L_k),$$

i.e., clauses containing $k \geq 4$ literals, into clauses with at most three literals, in such a way that satisfiability is preserved.

Example 10.8. For example, consider the following clause with $k = 6$ literals:

$$C = (L_1 \vee L_2 \vee L_3 \vee L_4 \vee L_5 \vee L_6).$$

We create 3 new boolean variables y_1, y_2, y_3, and the 4 clauses

$$(L_1 \vee L_2 \vee y_1),\ (\overline{y_1} \vee L_3 \vee y_2),\ (\overline{y_2} \vee L_4 \vee y_3),\ (\overline{y}_3 \vee L_5 \vee L_6).$$

Let C' be the conjunction of these clauses. We claim that C is satisfiable iff C' is.

Assume that C' is satisfiable but C is not. If so, in any truth assigment v, $v(L_i) = \mathbf{F}$, for $i = 1, 2, \ldots, 6$. To satisfy the first clause, we must have $v(y_1) = \mathbf{T}$., Then to satisfy the second clause, we must have $v(y_2) = \mathbf{T}$, and similarly satisfy the third clause, we must have $v(y_3) = \mathbf{T}$. However, since $v(L_5) = \mathbf{F}$ and $v(L_6) = \mathbf{F}$, the only way to satisfy the fourth clause is to have $v(y_3) = \mathbf{F}$, contradicting that $v(y_3) = \mathbf{T}$. Thus, C is indeed satisfiable.

Let us now assume that C is satisfiable. This means that there is a smallest index i such that L_i is satisfied.

Say $i = 1$, so $v(L_1) = \mathbf{T}$. Then if we let $v(y_1) = v(y_2) = v(y_3) = \mathbf{F}$, we see that C' is satisfied.

Say $i = 2$, so $v(L_1) = \mathbf{F}$ and $v(L_2) = \mathbf{T}$. Again if we let $v(y_1) = v(y_2) = v(y_3) = \mathbf{F}$, we see that C' is satisfied.

Say $i = 3$, so $v(L_1) = \mathbf{F}$, $v(L_2) = \mathbf{F}$, and $v(L_3) = \mathbf{T}$. If we let $v(y_1) = \mathbf{T}$ and $v(y_2) = v(y_3) = \mathbf{F}$, we see that C' is satisfied.

Say $i = 4$, so $v(L_1) = \mathbf{F}$, $v(L_2) = \mathbf{F}$, $v(L_3) = \mathbf{F}$, and $v(L_4) = \mathbf{T}$. If we let $v(y_1) = \mathbf{T}$, $v(y_2) = \mathbf{T}$ and $v(y_3) = \mathbf{F}$, we see that C' is satisfied.

Say $i = 5$, so $v(L_1) = \mathbf{F}$, $v(L_2) = \mathbf{F}$, $v(L_3) = \mathbf{F}$, $v(L_4) = \mathbf{F}$, and $v(L_5) = \mathbf{T}$. If we let $v(y_1) = \mathbf{T}$, $v(y_2) = \mathbf{T}$ and $v(y_3) = \mathbf{T}$, we see that C' is satisfied.

Say $i = 6$, so $v(L_1) = \mathbf{F}$, $v(L_2) = \mathbf{F}$, $v(L_3) = \mathbf{F}$, $v(L_4) = \mathbf{F}$, $v(L_5) = \mathbf{F}$, and $v(L_6) = \mathbf{T}$. Again, if we let $v(y_1) = \mathbf{T}$, $v(y_2) = \mathbf{T}$ and $v(y_3) = \mathbf{T}$, we see that C' is satisfied.

Therefore if C is satisfied, then C' is satisfied in all cases.

In general, for every long clause, create $k - 3$ new boolean variables $y_1, \ldots y_{k-3}$, and the $k - 2$ clauses

$$\begin{aligned}&(L_1 \vee L_2 \vee y_1), (\overline{y_1} \vee L_3 \vee y_2), (\overline{y_2} \vee L_4 \vee y_3), \cdots,\\ &(\overline{y}_{k-4} \vee L_{k-2} \vee y_{k-3}), (\overline{y}_{k-3} \vee L_{k-1} \vee L_k).\end{aligned}$$

Let C' be the conjunction of these clauses. We claim that C is satisfiable iff C' is.

Assume that C' is satisfiable, but that C is not. Then for every truth assignment v, we have $v(L_i) = \mathbf{F}$, for $i = 1, \ldots, k$.

However, C' is satisfied by some v, and the only way this can happen is that $v(y_1) = \mathbf{T}$, to satisfy the first clause. Then $v(\overline{y_1}) = \mathbf{F}$, and we must have $v(y_2) = \mathbf{T}$, to satisfy the second clause.

By induction, we must have $v(y_{k-3}) = \mathbf{T}$, to satisfy the next to the last clause. However, the last clause is now false, a contradiction.

Thus, if C' is satisfiable, then so is C.

Conversely, assume that C is satisfiable. If so, there is some truth assignment, v, so that $v(C) = \mathbf{T}$, and thus, there is a smallest index i, with $1 \leq i \leq k$, so that $v(L_i) = \mathbf{T}$ (and so, $v(L_j) = \mathbf{F}$ for all $j < i$).

Let v' be the assignment extending v defined so that

$$v'(y_j) = \mathbf{F} \quad \text{if} \quad \max\{1, i-1\} \leq j \leq k-3,$$

and $v'(y_j) = \mathbf{T}$, otherwise.

It is easily checked that $v'(C') = \mathbf{T}$. $\square$

Another version of 3-SAT can be considered, in which every clause has exactly three literals. We will call this the problem *exact* 3-SAT.

Theorem 10.10. *(Cook, 1971) The satisfiability problem for exact* 3-SAT *is $\mathcal{NP}$-complete.*

Proof. A clause of the form (L) is satisfiable iff the following four clauses are satisfiable:

$$(L \vee u \vee v), (L \vee \overline{u} \vee v), (L \vee u \vee \overline{v}), (L \vee \overline{u} \vee \overline{v})$$

where u, v are new variables. A clause of the form $(L_1 \vee L_2)$ is satisfiable iff the following two clauses are satisfiable:

$$(L_1 \vee L_2 \vee u), (L_1 \vee L_2 \vee \overline{u}).$$

Thus, we have a reduction of 3-SAT to exact 3-SAT. $\square$

We now make some remarks about the conversion of propositions to CNF and about the satisfiability of arbitrary propositions.

10.9 Satisfiability of Arbitrary Propositions and CNF

We begin by discussing how to convert an arbitrary proposition to conjunctive normal form. In general, given a proposition A, a proposition A' in CNF equivalent to A may have an exponential length in the size of A. Thus it is not obvious that the satisfiability problem for arbitrary propositions belongs to $\mathcal{NP}$. However, using new variables, there is an algorithm to convert a proposition A to another proposition A' (containing the new variables) whose length is polynomial in the length of A and such that A is satisfiable iff A' is satisfiable.

Thus the satisfiability problem for arbitrary propositions belongs to $\mathcal{NP}$. We also briefly discuss the issue of uniqueness of the CNF. In short, it is not unique!

Recall the definition of arbitrary propositions.

Definition 10.10. The set of *propositions* (over the connectives $\vee$, $\wedge$, and $\neg$) is defined inductively as follows:

(1) Every propositional letter, $x \in \mathbf{PV}$, is a proposition (an *atomic* proposition).

(2) If A is a proposition, then $\neg A$ is a proposition.

(3) If A and B are propositions, then $(A \vee B)$ is a proposition.

(4) If A and B are propositions, then $(A \wedge B)$ is a proposition.

Two propositions A and B are *equivalent*, denoted $A \equiv B$, if

$$v \models A \quad \text{iff} \quad v \models B$$

for all truth assignments, v. It is easy to show that $A \equiv B$ iff the proposition

$$(\neg A \vee B) \wedge (\neg B \vee A)$$

is valid.

Definition 10.11. A proposition P is in *conjunctive normal form* (*CNF*) if it is a conjunction $P = C_1 \wedge \cdots \wedge C_n$ of propositions C_j which are disjunctions of literals (a literal is either a variable x or the negation $\neg x$ (also denoted $\overline{x}$) of a variable x).

A proposition P is in *disjunctive normal form* (*DNF*) if it is a disjunction $P = D_1 \vee \cdots \vee D_n$ of propositions D_j which are conjunctions of literals.

There are propositions such that any equivalent proposition in CNF has size exponential in terms of the original proposition.

Example 10.9. Here is such an example:

$$A = (x_1 \wedge x_2) \vee (x_3 \wedge x_4) \vee \cdots \vee (x_{2n-1} \wedge x_{2n}).$$

Observe that it is in DNF. We will prove a little later that any CNF for A contains 2^n occurrences of variables.

Proposition 10.11. *Every proposition A is equivalent to a proposition A' in CNF.*

There are several ways of proving Proposition 10.11. One method is algebraic, and consists in using the algebraic laws of boolean algebra. First one may convert a proposition to *negation normal form*, or *nnf*.

Definition 10.12. A proposition is in *negation normal form* or *nnf* if all occurrences of $\neg$ only appear in front of propositional variables, but not in front of compound propositions.

Any proposition can be converted to an equivalent one in nnf by using the de Morgan laws:

$$\begin{aligned}\neg(A \vee B) &\equiv (\neg A \wedge \neg B)\\ \neg(A \wedge B) &\equiv (\neg A \vee \neg B)\\ \neg\neg A &\equiv A.\end{aligned}$$

Observe that if A has n connectives, then the equivalent formula A' in nnf has at most $2n-1$ connectives. Then a proposition in nnf can be converted to CNF,

A nice method to convert a proposition in nnf to CNF is to construct a tree whose nodes are labeled with sets of propositions using the following (Gentzen-style) rules:

$$\frac{P, \Delta \qquad Q, \Delta}{(P \wedge Q), \Delta}$$

and

$$\frac{P, Q, \Delta}{(P \vee Q), \Delta}$$

where Δ stands for any set of propositions (even empty), and the comma stands for union. Thus, it is assumed that $(P \wedge Q) \notin \Delta$ in the first case, and that $(P \vee Q) \notin \Delta$ in the second case.

Since we interpret a set, Γ, of propositions as a disjunction, a valuation, v, satisfies Γ iff it satisfies *some* proposition in Γ.

Observe that a valuation v satisfies the conclusion of a rule iff it satisfies both premises in the first case, and the single premise in the second case. Using these rules, we can build a finite tree whose leaves are labeled with sets of literals.

By the above observation, a valuation v satisfies the proposition labeling the root of the tree iff it satisfies all the propositions labeling the leaves of the tree.

But then, a CNF for the original proposition A (in nnf, at the root of the tree) is the conjunction of the clauses appearing as the leaves of the tree. We may exclude the clauses that are tautologies, and we may discover in the process that A is a tautology (when all leaves are tautologies).

Example 10.10. An illustration of the above method to convert the proposition

$$A = (x_1 \wedge y_1) \vee (x_2 \wedge y_2)$$

is shown below:

$$
\dfrac{\dfrac{x_1, x_2 \qquad x_1, y_2}{x_1, x_2 \wedge y_2} \qquad \dfrac{y_1, x_2 \qquad y_1, y_2}{y_1, x_2 \wedge y_2}}{\dfrac{x_1 \wedge y_1, x_2 \wedge y_2}{(x_1 \wedge y_1) \vee (x_2 \wedge y_2)}}
$$

We obtain the CNF

$$B = (x_1 \vee x_2) \wedge (x_1 \vee y_2) \wedge (y_1 \vee x_2) \wedge (y_1 \vee y_2).$$

Remark: Rules for dealing for $\neg$ can also be created. In this case, we work with pairs of sets of propositions,

$$\Gamma \rightarrow \Delta,$$

where, the propositions in Γ are interpreted conjunctively, and the propositions in Δ are interpreted disjunctively. We obtain a sound and complete proof system for propositional logic (a "Gentzen-style" proof system, see *Logic for Computer Science*, Gallier [19]).

Going back to our "bad" proposition A from Example 10.9, by induction, we see that any tree for A has 2^n leaves.

However, the following result holds.

Proposition 10.12. *For any proposition A, we can construct in polynomial time a formula A' in CNF, so that A is satisfiable iff A' is satisfiable, by creating new variables.*

Sketch of proof. First we convert A to nnf, which yields a proposition at most twice as long. Then we proceed recursively. For a conjunction $C \wedge D$, we apply recursively the procedure to C and D. The trick is that for a disjunction $C \vee D$, first we apply recursively the procedure to C and D obtain

$$(C_1 \wedge \cdots \wedge C_m) \vee (D_1 \wedge \cdots \wedge D_n)$$

where the C_i's and the D_j's are clauses. Then we create

$$(C_1 \vee y) \wedge \cdots \wedge (C_m \vee y) \wedge (D_1 \vee \overline{y}) \wedge \cdots \wedge (D_n \vee \overline{y}),$$

where y is a new variable.

It can be shown that the number of new variables required is at most quadratic in the size of A. For details on this construction see Hopcroft, Motwani and Ullman [29] (Section 10.3.3), but beware that the proof on page 455 contains a mistake. Repair the mistake. $\square$

Example 10.11. Consider the proposition

$$A = (x_1 \wedge \neg x_2) \vee ((\neg x_1 \wedge x_2) \vee (x_2 \vee x_3)).$$

First, since x_1 and $\neg x_2$ are clauses, we get

$$A_1 = x_1 \wedge \neg x_2.$$

Since $\neg x_1$, x_2 and $x_2 \vee x_3$ are clauses, from $(\neg x_1 \wedge x_2) \vee (x_2 \vee x_3)$ we construct

$$A_2 = (\neg x_1 \vee y_1) \wedge (x_2 \vee y_1) \wedge (x_2 \vee x_3 \vee \neg y_1).$$

Next, since A_1 and A_2 are conjunctions of clauses, we construct

$$A' = (x_1 \vee y_2) \wedge (\neg x_2 \vee y_2) \wedge (\neg x_1 \vee y_1 \vee \neg y_2) \wedge (x_2 \vee y_1 \vee \neg y_2) \wedge (x_2 \vee x_3 \vee \neg y_1 \vee \neg y_2),$$

a conjunction of clauses which is satisfiable iff A is satisfiable.

Warning: In general, the proposition A' is *not* equivalent to the proposition A.

The question of uniqueness of the CNF is a bit tricky. For example, the proposition

$$A = (u \wedge (x \vee y)) \vee (\neg u \wedge (x \vee y))$$

has

$$\begin{aligned} A_1 &= (u \vee x \vee y) \wedge (\neg u \vee x \vee y) \\ A_2 &= (u \vee \neg u) \wedge (x \vee y) \\ A_3 &= x \vee y, \end{aligned}$$

as equivalent propositions in CNF!

We can get a *unique* CNF equivalent to a given proposition if we do the following:

(1) Let $\mathrm{Var}(A) = \{x_1, \ldots, x_m\}$ be the set of variables occurring in A.

(2) Define a *maxterm w.r.t.* $\mathrm{Var}(A)$ as any disjunction of m pairwise distinct literals formed from $\mathrm{Var}(A)$, and not containing both some variable x_i and its negation $\neg x_i$.

(3) Then it can be shown that for any proposition A that is not a tautology, there is a *unique* proposition in CNF *equivalent* to A, whose clauses consist of maxterms formed from $\mathrm{Var}(A)$.

The above definition can yield strange results. For instance, the CNF of any unsatisfiable proposition with m distinct variables is the conjunction of all of its 2^m maxterms! The above notion does not cope well with minimality.

For example, according to the above, the CNF of

$$A = (u \wedge (x \vee y)) \vee (\neg u \wedge (x \vee y))$$

should be

$$A_1 = (u \vee x \vee y) \wedge (\neg u \vee x \vee y).$$

Chapter 11

Some $\mathcal{NP}$-Complete Problems

11.1 Statements of the Problems

In this chapter we will show that certain classical algorithmic problems are $\mathcal{NP}$-complete. This chapter is heavily inspired by Lewis and Papadimitriou's excellent treatment [35]. In order to study the complexity of these problems in terms of resource (time or space) bounded Turing machines (or RAM programs), it is crucial to be able to encode instances of a problem P as strings in a language L_P. Then an instance of a problem P is solvable iff the corresponding string belongs to the language L_P. This implies that our problems must have a yes–no answer, which is not always the usual formulation of optimization problems where what is required is to find some *optimal* solution, that is, a solution minimizing or maximizing so objective (cost) function F. For example the standard formulation of the traveling salesman problem asks for a tour (of the cities) of minimal cost.

Fortunately, there is a trick to reformulate an optimization problem as a yes–no answer problem, which is to explicitly incorporate a *budget* (or *cost*) term B into the problem, and instead of asking whether some objective function F has a minimum or a maximum w, we ask whether there is a solution w such that $F(w) \leq B$ in the case of a minimum solution, or $F(w) \geq B$ in the case of a maximum solution.

If we are looking for a minimum of F, we try to guess the minimum value B of F and then we solve the problem of finding w such that $F(w) \leq B$. If our guess for B is too small, then we fail. In this case, we try again with a larger value of B. Otherwise, if B was not too small we find some w such that $F(w) \leq B$, but w may not correspond to a minimum of F, so we try again with a smaller value of B, and so on. This yields an approximation method to find a minimum of F.

Similarly, if we are looking for a maximum of F, we try to guess the maximum value B of F and then we solve the problem of finding w such that $F(w) \geq B$. If our guess for B is too large, then we fail. In this case, we try again with a smaller value of B. Otherwise, if B was not too large we find some w such that $F(w) \geq B$, but w may not correspond to a maximum of F, so we try again with a greater value of B, and so on. This yields an approximation method to find a maximum of F.

We will see several examples of this technique in Problems 5–8 listed below.

The problems that will consider are

(1) Exact Cover

(2) Hamiltonian Cycle for directed graphs

(3) Hamiltonian Cycle for undirected graphs

(4) The Traveling Salesman Problem

(5) Independent Set

(6) Clique

(7) Node Cover

(8) Knapsack, also called subset sum

(9) Inequivalence of $*$-free Regular Expressions

(10) The 0-1-integer programming problem

We begin by describing each of these problems.

(1) **Exact Cover**

We are given a finite nonempty set $U = \{u_1, \ldots, u_n\}$ (the universe), and a family $\mathcal{F} = \{S_1, \ldots, S_m\}$ of $m \geq 1$ nonempty subsets of U. The question is whether there is an *exact cover*, that is, a subfamily $\mathcal{C} \subseteq \mathcal{F}$ of subsets in $\mathcal{F}$ such that the sets in $\mathcal{C}$ are disjoint and their union is equal to U.

For example, let $U = \{u_1, u_2, u_3, u_4, u_5, u_6\}$, and let $\mathcal{F}$ be the family

$$\mathcal{F} = \{\{u_1, u_3\}, \{u_2, u_3, u_6\}, \{u_1, u_5\}, \{u_2, u_3, u_4\}, \{u_5, u_6\}, \{u_2, u_4\}\}.$$

The subfamily

$$\mathcal{C} = \{\{u_1, u_3\}, \{u_5, u_6\}, \{u_2, u_4\}\}$$

is an exact cover.

It is easy to see that **Exact Cover** is in $\mathcal{NP}$. To prove that it is $\mathcal{NP}$-complete, we will reduce the **Satisfiability Problem** to it. This means that we provide a method running in polynomial time that converts every instance of the **Satisfiability Problem** to an instance of **Exact Cover**, such that the first problem has a solution iff the converted problem has a solution.

(2) **Hamiltonian Cycle (for Directed Graphs)**

Recall that a *directed graph* G is a pair $G = (V, E)$, where $E \subseteq V \times V$. Elements of V are called *nodes* (or *vertices*). A pair $(u, v) \in E$ is called an *edge* of G. We will restrict ourselves to *simple graphs*, that is, graphs without edges of the form (u, u); equivalently, $G = (V, E)$ is a simple graph if whenever $(u, v) \in E$, then $u \neq v$.

Given any two nodes $u, v \in V$, a *path from u to v* is any sequence of $n+1$ edges ($n \geq 0$)

$$(u, v_1), (v_1, v_2), \ldots, (v_n, v).$$

(If $n = 0$, a path from u to v is simply a single edge, (u, v).)

A directed graph G is *strongly connected* if for every pair $(u, v) \in V \times V$, there is a path from u to v. A *closed path, or cycle*, is a path from some node u to itself. We will restrict out attention to finite graphs, i.e. graphs (V, E) where V is a finite set.

Definition 11.1. Given a directed graph G, a *Hamiltonian cycle* is a cycle that passes through all the nodes exactly once (note, some edges may not be traversed at all).

Hamiltonian Cycle Problem (for Directed Graphs): Given a directed graph G, is there an Hamiltonian cycle in G?

Is there is a Hamiltonian cycle in the directed graph D shown in Figure 11.1?

Finding a Hamiltonian cycle in this graph does not appear to be so easy! A solution is shown in Figure 11.2 below.

It is easy to see that **Hamiltonian Cycle (for Directed Graphs)** is in $\mathcal{NP}$. To prove that it is $\mathcal{NP}$-complete, we will reduce **Exact Cover** to it. This means that we provide a method running in polynomial time that converts every instance of **Exact Cover** to an instance of **Hamiltonian Cycle (for Directed Graphs)** such that the first problem has a solution iff the converted problem has a solution. This is perphaps the hardest reduction.

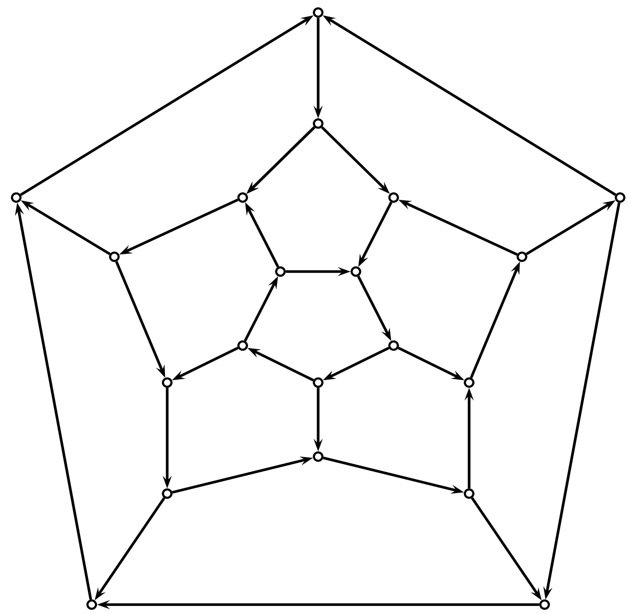


Figure 11.1: A tour "around the world."

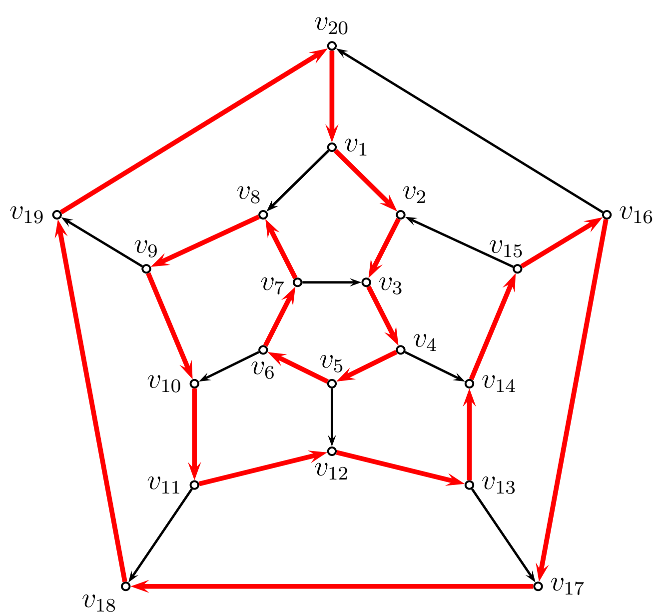


Figure 11.2: A Hamiltonian cycle in D.

(3) **Hamiltonian Cycle (for Undirected Graphs)**

Recall that an *undirected graph* G is a pair $G = (V, E)$, where E is a set of subsets $\{u, v\}$ of V consisting of exactly two distinct elements. Elements of V are called *nodes* (or *vertices*). A pair $\{u, v\} \in E$ is called an *edge* of G.

Given any two nodes $u, v \in V$, a *path from u to v* is any sequence of n nodes ($n \geq 2$)

$$u = u_1, u_2, \ldots, u_n = v$$

such that $\{u_i, u_{i+1}\} \in E$ for $i = 1, \ldots, n-1$. (If $n = 2$, a path from u to v is simply a single edge, $\{u, v\}$.)

An undirected graph G is *connected* if for every pair $(u, v) \in V \times V$, there is a path from u to v. A *closed path, or cycle*, is a path from some node u to itself.

Definition 11.2. Given an undirected graph G, a *Hamiltonian cycle* is a cycle that passes through all the nodes exactly once (note, some edges may not be traversed at all).

Hamiltonian Cycle Problem (for Undirected Graphs): Given an undirected graph G, is there an Hamiltonian cycle in G?

An instance of this problem is obtained by changing every directed edge in the directed graph of Figure 11.1 to an undirected edge. The directed Hamiltonian cycle given in Figure 11.1 is also an undirected Hamiltonian cycle of the undirected graph of Figure 11.3.

We see immediately that **Hamiltonian Cycle (for Undirected Graphs)** is in $\mathcal{NP}$. To prove that it is $\mathcal{NP}$-complete, we will reduce **Hamiltonian Cycle (for Directed Graphs)** to it. This means that we provide a method running in polynomial time that converts every instance of **Hamiltonian Cycle (for Directed Graphs)** to an instance of **Hamiltonian Cycle (for Undirected Graphs)** such that the first problem has a solution iff the converted problem has a solution. This is an easy reduction.

(4) **Traveling Salesman Problem**

We are given a set $\{c_1, c_2, \ldots, c_n\}$ of $n \geq 2$ cities, and an $n \times n$ matrix $D = (d_{ij})$ of nonnegative integers, where d_{ij} is the *distance* (or *cost*) of traveling from city c_i to city c_j. We assume that $d_{ii} = 0$ and $d_{ij} = d_{ji}$ for all i, j, so that the matrix D is symmetric and has zero diagonal.

Traveling Salesman Problem: Given some $n \times n$ matrix $D = (d_{ij})$ as above and some integer $B \geq 0$ (the *budget* of the traveling salesman), find a permutation π of $\{1, 2, \ldots, n\}$ such that

$$c(\pi) = d_{\pi(1)\pi(2)} + d_{\pi(2)\pi(3)} + \cdots + d_{\pi(n-1)\pi(n)} + d_{\pi(n)\pi(1)} \leq B.$$

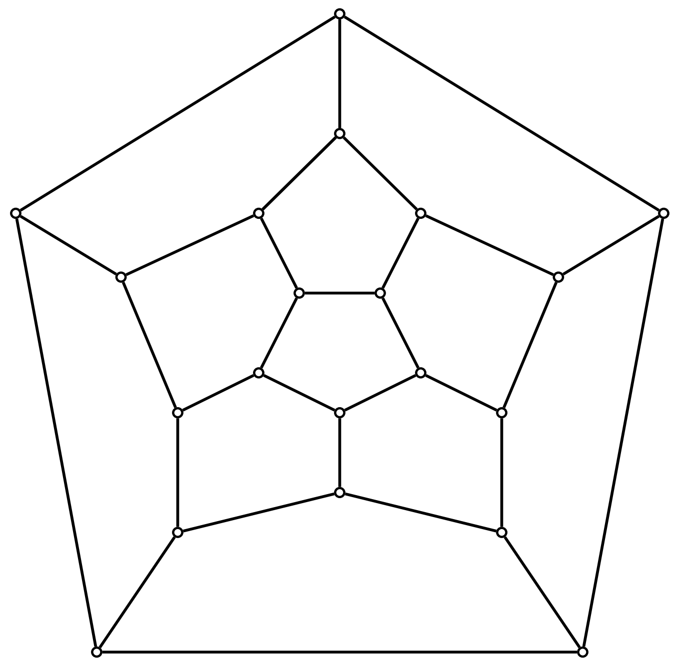


Figure 11.3: A tour "around the world," undirected version.

The quantity $c(\pi)$ is the *cost* of the trip specified by π. The Traveling Salesman Problem has been stated in terms of a budget so that it has a yes or no answer, which allows us to convert it into a language. A minimal solution corresponds to the smallest feasible value of B.

Example 11.1. Consider the 4×4 symmetric matrix given by

$$D = \begin{pmatrix} 0 & 2 & 1 & 1 \\ 2 & 0 & 1 & 1 \\ 1 & 1 & 0 & 3 \\ 1 & 1 & 3 & 0 \end{pmatrix},$$

and the budget $B = 4$. The tour specified by the permutation

$$\pi = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 1 & 4 & 2 & 3 \end{pmatrix}$$

has cost 4, since

$$\begin{aligned} c(\pi) &= d_{\pi(1)\pi(2)} + d_{\pi(2)\pi(3)} + d_{\pi(3)\pi(4)} + d_{\pi(4)\pi(1)} \\ &= d_{14} + d_{42} + d_{23} + d_{31} \\ &= 1 + 1 + 1 + 1 = 4. \end{aligned}$$

The cities in this tour are traversed in the order

$$(1, 4, 2, 3, 1).$$

It is clear that the **Traveling Salesman Problem** is in $\mathcal{NP}$. To show that it is $\mathcal{NP}$-complete, we reduce the **Hamiltonian Cycle Problem (Undirected Graphs)** to it. This means that we provide a method running in polynomial time that converts every instance of **Hamiltonian Cycle Problem (Undirected Graphs)** to an instance of the **Traveling Salesman Problem** such that the first problem has a solution iff the converted problem has a solution.

(5) **Independent Set**

The problem is this: Given an undirected graph $G = (V, E)$ and an integer $K \geq 2$, is there a set C of nodes with $|C| \geq K$ such that for all $v_i, v_j \in C$, there is *no* edge $\{v_i, v_j\} \in E$?

A maximal independent set with 3 nodes is shown in Figure 11.4. A maximal solution

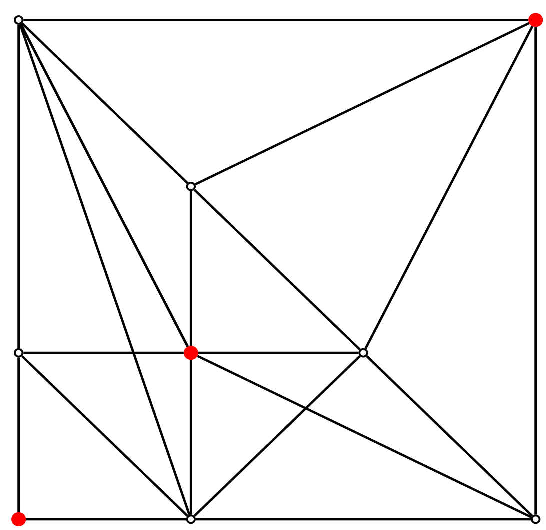


Figure 11.4: A maximal Independent Set in a graph.

corresponds to the largest feasible value of K. The problem **Independent Set** is obviously in $\mathcal{NP}$. To show that it is $\mathcal{NP}$-complete, we reduce **Exact 3-Satisfiability** to it. This means that we provide a method running in polynomial time that converts every instance of **Exact 3-Satisfiability** to an instance of **Independent Set** such that the first problem has a solution iff the converted problem has a solution.

(6) **Clique**

The problem is this: Given an undirected graph $G = (V, E)$ and an integer $K \geq 2$, is there a set C of nodes with $|C| \geq K$ such that for all $v_i, v_j \in C$, there is *some* edge $\{v_i, v_j\} \in E$? Equivalently, does G contain a complete subgraph with at least K nodes?

A maximal clique with 4 nodes is shown in Figure 11.5. A maximal solution corresponds

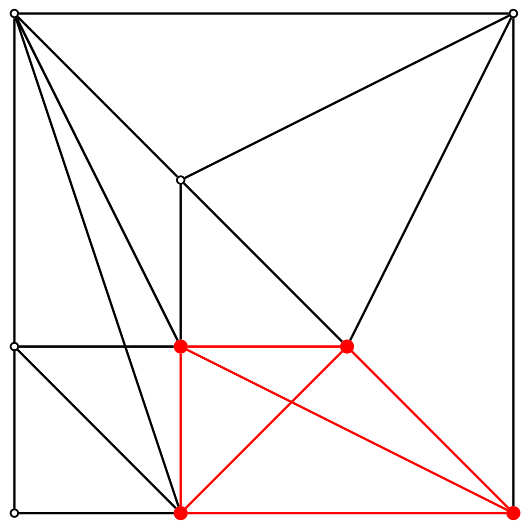


Figure 11.5: A maximal Clique in a graph.

to the largest feasible value of K. The problem **Clique** is obviously in $\mathcal{NP}$. To show that it is $\mathcal{NP}$-complete, we reduce **Independent Set** to it. This means that we provide a method running in polynomial time that converts every instance of **Independent Set** to an instance of **Clique** such that the first problem has a solution iff the converted problem has a solution.

(7) **Node Cover**

The problem is this: Given an undirected graph $G = (V, E)$ and an integer $B \geq 2$, is there a set C of nodes with $|C| \leq B$ such that C covers all edges in G, which means that for every edge $\{v_i, v_j\} \in E$, either $v_i \in C$ or $v_j \in C$?

A minimal node cover with 6 nodes is shown in Figure 11.6. A minimal solution corresponds to the smallest feasible value of B. The problem **Node Cover** is obviously in $\mathcal{NP}$. To show that it is $\mathcal{NP}$-complete, we reduce **Independent Set** to it. This means that we provide a method running in polynomial time that converts every instance of

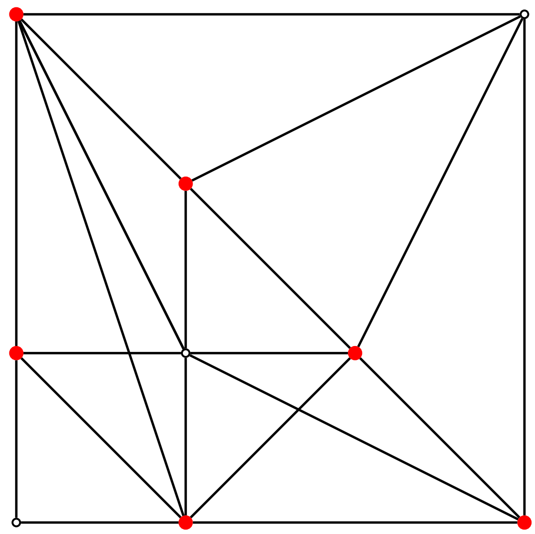


Figure 11.6: A minimal Node Cover in a graph.

Independent Set to an instance of **Node Cover** such that the first problem has a solution iff the converted problem has a solution.

The Node Cover problem has the following interesting interpretation: think of the nodes of the graph as rooms of a museum (or art gallery *etc.*), and each edge as a straight corridor that joins two rooms. Then Node Cover may be useful in assigning as few as possible guards to the rooms, so that all corridors can be seen by a guard.

(8) **Knapsack (also called Subset sum)**

The problem is this: Given a finite nonempty set $S = \{a_1, a_2, \ldots, a_n\}$ of nonnegative integers, and some integer $K \geq 0$, all represented in binary, is there a nonempty subset $I \subseteq \{1, 2, \ldots, n\}$ such that

$$\sum_{i \in I} a_i = K?$$

A "concrete" realization of this problem is that of a hiker who is trying to fill her/his backpack to its maximum capacity with items of varying weights or values.

It is easy to see that the **Knapsack** Problem is in $\mathcal{NP}$. To show that it is $\mathcal{NP}$-complete, we reduce **Exact Cover** to it. This means that we provide a method running in polynomial time that converts every instance of **Exact Cover** to an instance of **Knapsack** Problem such that the first problem has a solution iff the converted problem has a solution.

Remark: The **0-1 Knapsack Problem** is defined as the following problem. Given a set of n items, numbered from 1 to n, each with a weight $w_i \in \mathbb{N}$ and a value $v_i \in \mathbb{N}$, given a maximum capacity $W \in \mathbb{N}$ and a budget $B \in \mathbb{N}$, is there a set of n variables $x_1, \ldots, x_n$ with $x_i \in \{0, 1\}$ such that

$$\sum_{i=1}^{n} x_i v_i \geq B,$$
$$\sum_{i=1}^{n} x_i w_i \leq W.$$

Informally, the problem is to pick items to include in the knapsack so that the sum of the values exceeds a given minimum B (the goal is to maximize this sum), and the sum of the weights is less than or equal to the capacity W of the knapsack. A maximal solution corresponds to the largest feasible value of B.

The **Knapsack** Problem as we defined it (which is how Lewis and Papadimitriou define it) is the special case where $v_i = w_i = 1$ for $i = 1, \ldots, n$ and $W = B$. For this reason, it is also called the **Subset Sum** Problem. Clearly, the **Knapsack** (**Subset Sum**) Problem reduces to the **0-1 Knapsack** Problem, and thus the **0-1 Knapsack** Problem is also NP-complete.

(9) **Inequivalence of $*$-free Regular Expressions**

Recall that the problem of deciding the equivalence $R_1 \cong R_2$ of two regular expressions R_1 and R_2 is the problem of deciding whether R_1 and R_2 define the same language, that is, $\mathcal{L}[R_1] = \mathcal{L}[R_2]$. Is this problem in $\mathcal{NP}$?

In order to show that the equivalence problem for regular expressions is in $\mathcal{NP}$ we would have to be able to somehow check in polynomial time that two expressions define the same language, but this is still an open problem.

What might be easier is to decide whether two regular expressions R_1 and R_2 are *inequivalent*. For this, we just have to find a string w such that either $w \in \mathcal{L}[R_1] - \mathcal{L}[R_2]$ or $w \in \mathcal{L}[R_2] - \mathcal{L}[R_1]$. The problem is that if we can guess such a string w, we still have to check in polynomial time that $w \in (\mathcal{L}[R_1] - \mathcal{L}[R_2]) \cup (\mathcal{L}[R_2] - \mathcal{L}[R_1])$, and this implies that there is a bound on the length of w which is polynomial in the sizes of R_1 and R_2. Again, this is an open problem.

To obtain a problem in $\mathcal{NP}$ we have to consider a restricted type of regular expressions, and it turns out that $*$-free regular expressions are the right candidate. A *$*$-free regular expression* is a regular expression which is built up from the atomic expressions using only $+$ and $\cdot$, but not $*$. For example,

$$R = ((a + b)aa(a + b) + aba(a + b)b)$$

is such an expression.

It is easy to see that if R is a $*$-free regular expression, then for every string $w \in \mathcal{L}[R]$ we have $|w| \leq |R|$. In particular, $\mathcal{L}[R]$ is finite. The above observation shows that if R_1 and R_2 are $*$-free and if there is a string $w \in (\mathcal{L}[R_1] - \mathcal{L}[R_2]) \cup (\mathcal{L}[R_2] - \mathcal{L}[R_1])$, then $|w| \leq |R_1| + |R_2|$, so we can indeed check this in polynomial time. It follows that the inequivalence problem for $*$ -free regular expressions is in $\mathcal{NP}$. To show that it is $\mathcal{NP}$-complete, we reduce the **Satisfiability Problem** to it. This means that we provide a method running in polynomial time that converts every instance of **Satisfiability Problem** to an instance of **Inequivalence of Regular Expressions** such that the first problem has a solution iff the converted problem has a solution.

Observe that both problems of **Inequivalence of Regular Expressions** and **Equivalence of Regular Expressions** are as hard as **Inequivalence of $*$-free Regular Expressions**, since if we could solve the first two problems in polynomial time, then we we could solve **Inequivalence of $*$-free Regular Expressions** in polynomial time, but since this problem is $\mathcal{NP}$-complete, we would have $\mathcal{P} = \mathcal{NP}$. This is very unlikely, so the complexity of **Equivalence of Regular Expressions** remains open.

(10) 0-1 **integer programming problem**

Let A be any $p \times q$ matrix with integer coefficients and let $b \in \mathbb{Z}^p$ be any vector with integer coefficients. The 0-1 **integer programming problem** is to find whether a system of p linear equations in q variables

$$\begin{aligned} a_{11}x_1 + \cdots + a_{1q}x_q &= b_1 \\ \vdots \qquad\quad & \quad \vdots \\ a_{i1}x_1 + \cdots + a_{iq}x_q &= b_i \\ \vdots \qquad\quad & \quad \vdots \\ a_{p1}x_1 + \cdots + a_{pq}x_q &= b_p \end{aligned}$$

with $a_{ij}, b_i \in \mathbb{Z}$ has any solution $x \in \{0,1\}^q$, that is, with $x_i \in \{0,1\}$. In matrix form, if we let

$$A = \begin{pmatrix} a_{11} & \cdots & a_{1q} \\ \vdots & \ddots & \vdots \\ a_{p1} & \cdots & a_{pq} \end{pmatrix}, \quad b = \begin{pmatrix} b_1 \\ \vdots \\ b_p \end{pmatrix}, \quad x = \begin{pmatrix} x_1 \\ \vdots \\ x_q \end{pmatrix},$$

then we write the above system as

$$Ax = b.$$

Example 11.2. Is there a solution $x = (x_1, x_2, x_3, x_4, x_5, x_6)$ of the linear system

$$\begin{pmatrix} 1 & -2 & 1 & 3 & -1 & 4 \\ 2 & 2 & -1 & 0 & 1 & -1 \\ -1 & 1 & 2 & 3 & -2 & 3 \\ 3 & 1 & -1 & 2 & -1 & 4 \\ 0 & 1 & -1 & 1 & 1 & 2 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{pmatrix} = \begin{pmatrix} 9 \\ 0 \\ 7 \\ 8 \\ 2 \end{pmatrix}$$

with $x_i \in \{0, 1\}$?

Indeed, $x = (1, 0, 1, 1, 0, 1)$ is a solution.

It is immediate that 0-1 **integer programming problem** is in $\mathcal{NP}$. To prove that it is $\mathcal{NP}$-complete we reduce the **bounded tiling** problem to it. This means that we provide a method running in polynomial time that converts every instance of the **bounded tiling** problem to an instance of the 0-1 **integer programming problem** such that the first problem has a solution iff the converted problem has a solution.

11.2 Proofs of $\mathcal{NP}$-Completeness

(1) **Exact Cover**

To prove that **Exact Cover** is $\mathcal{NP}$-complete, we reduce the **Satisfiability Problem** to it:

Satisfiability Problem $\leq_P$ **Exact Cover**

Given a set $F = \{C_1, \ldots, C_\ell\}$ of ℓ clauses constructed from n propositional variables $x_1, \ldots, x_n$, we must construct in polynomial time (in the sum of the lengths of the clauses) an instance $\tau(F) = (U, \mathcal{F})$ of **Exact Cover** such that F is satisfiable iff $\tau(F)$ has a solution.

Example 11.3. If

$$F = \{C_1 = (x_1 \vee \overline{x_2}),\ C_2 = (\overline{x_1} \vee x_2 \vee x_3),\ C_3 = (x_2),\ C_4 = (\overline{x_2} \vee \overline{x_3})\},$$

then the universe U is given by

$$U = \{x_1, x_2, x_3, C_1, C_2, C_3, C_4, p_{11}, p_{12}, p_{21}, p_{22}, p_{23}, p_{31}, p_{41}, p_{42}\},$$

and the family $\mathcal{F}$ consists of the subsets

$$
\begin{aligned}
&\{p_{11}\}, \{p_{12}\}, \{p_{21}\}, \{p_{22}\}, \{p_{23}\}, \{p_{31}\}, \{p_{41}\}, \{p_{42}\}\\
&T_{1,\mathbf{F}} = \{x_1, p_{11}\}\\
&T_{1,\mathbf{T}} = \{x_1, p_{21}\}\\
&T_{2,\mathbf{F}} = \{x_2, p_{22}, p_{31}\}\\
&T_{2,\mathbf{T}} = \{x_2, p_{12}, p_{41}\}\\
&T_{3,\mathbf{F}} = \{x_3, p_{23}\}\\
&T_{3,\mathbf{T}} = \{x_3, p_{42}\}\\
&\{C_1, p_{11}\}, \{C_1, p_{12}\}, \{C_2, p_{21}\}, \{C_2, p_{22}\}, \{C_2, p_{23}\},\\
&\{C_3, p_{31}\}, \{C_4, p_{41}\}, \{C_4, p_{42}\}.
\end{aligned}
$$

The above construction is illustrated in Figure 11.7.

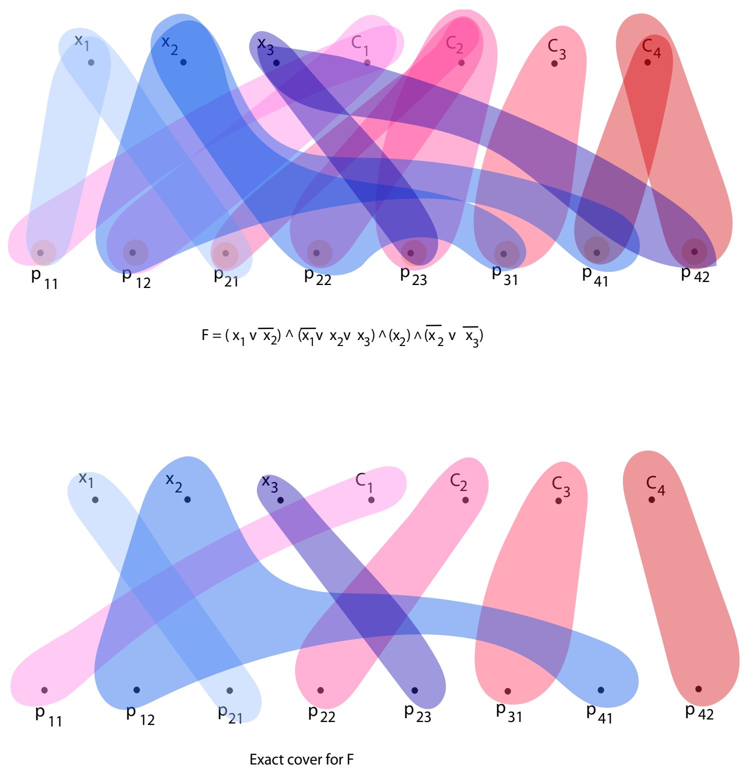


Figure 11.7: Construction of an exact cover from the set of clauses in Example 11.3.

It is easy to check that the set $\mathcal{C}$ consisting of the following subsets is an exact cover:

$$
\begin{aligned}
&T_{1,\mathbf{T}} = \{x_1, p_{21}\}, T_{2,\mathbf{T}} = \{x_2, p_{12}, p_{41}\}, T_{3,\mathbf{F}} = \{x_3, p_{23}\},\\
&\{C_1, p_{11}\}, \{C_2, p_{22}\}, \{C_3, p_{31}\}, \{C_4, p_{42}\}.
\end{aligned}
$$

The general method to construct $(U, \mathcal{F})$ from $F = \{C_1, \ldots, C_\ell\}$ proceeds as follows. The size n of the input is the sum of the lengths of the clauses C_i as strings, Say

$$C_j = (L_{j1} \vee \cdots \vee L_{jm_j})$$

is the jth clause in F, where L_{jk} denotes the kth literal in C_j and $m_j \geq 1$. The universe of $\tau(F)$ is the set

$$U = \{x_i \mid 1 \leq i \leq n\} \cup \{C_j \mid 1 \leq j \leq \ell\} \cup \{p_{jk} \mid 1 \leq j \leq \ell,\ 1 \leq k \leq m_j\}$$

where in the third set p_{jk} corresponds to the kth literal in C_j. The universe U can be constructed in time $O(n^2)$.

The following subsets are included in $\mathcal{F}$:

(a) There is a set $\{p_{jk}\}$ for every p_{jk}.

(b) For every boolean variable x_i, the following two sets are in $\mathcal{F}$:

$$T_{i,\mathbf{T}} = \{x_i\} \cup \{p_{jk} \mid L_{jk} = \overline{x_i}\}$$

which contains x_i and all negative occurrences of x_i, and

$$T_{i,\mathbf{F}} = \{x_i\} \cup \{p_{jk} \mid L_{jk} = x_i\}$$

which contains x_i and all its positive occurrences. Note carefully that $T_{i,\mathbf{T}}$ involves negative occurrences of x_i whereas $T_{i,\mathbf{F}}$ involves positive occurrences of x_i.

(c) For every clause C_j, the m_j sets $\{C_j, p_{jk}\}$ are in $\mathcal{F}$.

The subsets in (a), (b), (c) can be constructed in time $O(n^3)$. It remains to prove that F is satisfiable iff $\tau(F)$ has a solution. We claim that if v is a truth assignement that satisfies F, then we can make an exact cover $\mathcal{C}$ as follows:

For each x_i, we put the subset $T_{i,\mathbf{T}}$ in $\mathcal{C}$ iff $v(x_i) = \mathbf{T}$, else we we put the subset $T_{i,\mathbf{F}}$ in $\mathcal{C}$ iff $v(x_i) = \mathbf{F}$. Also, for every clause C_j, we put some subset $\{C_j, p_{jk}\}$ in $\mathcal{C}$ for a literal L_{jk} which is made true by v. By construction of $T_{i,\mathbf{T}}$ and $T_{i,\mathbf{F}}$, this p_{jk} is not in any set in $\mathcal{C}$ selected so far. Since by hypothesis F is satisfiable, such a literal exists for every clause. Having covered all x_i and C_j, we put a set $\{p_{jk}\}$ in $\mathcal{C}$ for every remaining p_{jk} which has not yet been covered by the sets already in $\mathcal{C}$.

Going back to Example 11.3, the truth assignment $v(x_1) = \mathbf{T}, v(x_2) = \mathbf{T}, v(x_3) = \mathbf{F}$ satisfies F, so we put

$$\begin{aligned} &T_{1,\mathbf{T}} = \{x_1, p_{21}\}, T_{2,\mathbf{T}} = \{x_2, p_{12}, p_{41}\}, T_{3,\mathbf{F}} = \{x_3, p_{23}\}, \\ &\{C_1, p_{11}\}, \{C_2, p_{22}\}, \{C_3, p_{31}\}, \{C_4, p_{42}\} \end{aligned}$$

in $\mathcal{C}$.

We leave as an exercise to check that the above procedure works.

Conversely, if $\mathcal{C}$ is an exact cover of $\tau(F)$, we define a truth assigment as follows:

For every x_i, if $T_{i,\mathbf{T}}$ is in $\mathcal{C}$, then we set $v(x_i) = \mathbf{T}$, else if $T_{i,\mathbf{F}}$ is in $\mathcal{C}$, then we set $v(x_i) = \mathbf{F}$. We leave it as an exercise to check that this procedure works.

Example 11.4. Given the exact cover

$$
\begin{aligned}
&T_{1,\mathbf{T}} = \{x_1, p_{21}\}, T_{2,\mathbf{T}} = \{x_2, p_{12}, p_{41}\}, T_{3,\mathbf{F}} = \{x_3, p_{23}\},\\
&\{C_1, p_{11}\}, \{C_2, p_{22}\}, \{C_3, p_{31}\}, \{C_4, p_{42}\},
\end{aligned}
$$

we get the satisfying assigment $v(x_1) = \mathbf{T}, v(x_2) = \mathbf{T}, v(x_3) = \mathbf{F}$.

If we now consider the proposition is CNF given by

$$F_2 = \{C_1 = (x_1 \lor \overline{x_2}),\ C_2 = (\overline{x_1} \lor x_2 \lor x_3),\ C_3 = (x_2),\ C_4 = (\overline{x_2} \lor \overline{x_3} \lor x_4)\}$$

where we have added the boolean variable x_4 to clause C_4, then U also contains x_4 and p_{43} so we need to add the following subsets to $\mathcal{F}$:

$$T_{4,\mathbf{F}} = \{x_4, p_{43}\},\ T_{4,\mathbf{T}} = \{x_4\},\ \{C_4, p_{43}\},\ \{p_{43}\}.$$

The truth assigment $v(x_1) = \mathbf{T}, v(x_2) = \mathbf{T}, v(x_3) = \mathbf{F}, v(x_4) = \mathbf{T}$ satisfies F_2, so an exact cover $\mathcal{C}$ is

$$
\begin{aligned}
&T_{1,\mathbf{T}} = \{x_1, p_{21}\}, T_{2,\mathbf{T}} = \{x_2, p_{12}, p_{41}\}, T_{3,\mathbf{F}} = \{x_3, p_{23}\}, T_{4,\mathbf{T}} = \{x_4\},\\
&\{C_1, p_{11}\}, \{C_2, p_{22}\}, \{C_3, p_{31}\}, \{C_4, p_{42}\}, \{p_{43}\}.
\end{aligned}
$$

The above construction is illustrated in Figure 11.8.

Observe that this time, because the truth assignment v makes both literals corresponding to p_{42} and p_{43} true and since we picked p_{42} to form the subset $\{C_4, p_{42}\}$, we need to add the singleton $\{p_{43}\}$ to $\mathcal{C}$ to cover all elements of U.

(2) **Hamiltonian Cycle (for Directed Graphs)**

To prove that **Hamiltonian Cycle (for Directed Graphs)** is $\mathcal{NP}$-complete, we will reduce **Exact Cover** to it:

Exact Cover $\leq_P$ **Hamiltonian Cycle (for Directed Graphs)**

We need to find an algorithm working in polynomial time that converts an instance $(U, \mathcal{F})$ of **Exact Cover** to a directed graph $G = \tau(U, \mathcal{F})$ such that G has a Hamiltonian cycle iff $(U, \mathcal{F})$ has an exact cover. The size n of the input $(U, \mathcal{F})$ is $|U| + |\mathcal{F}|$.

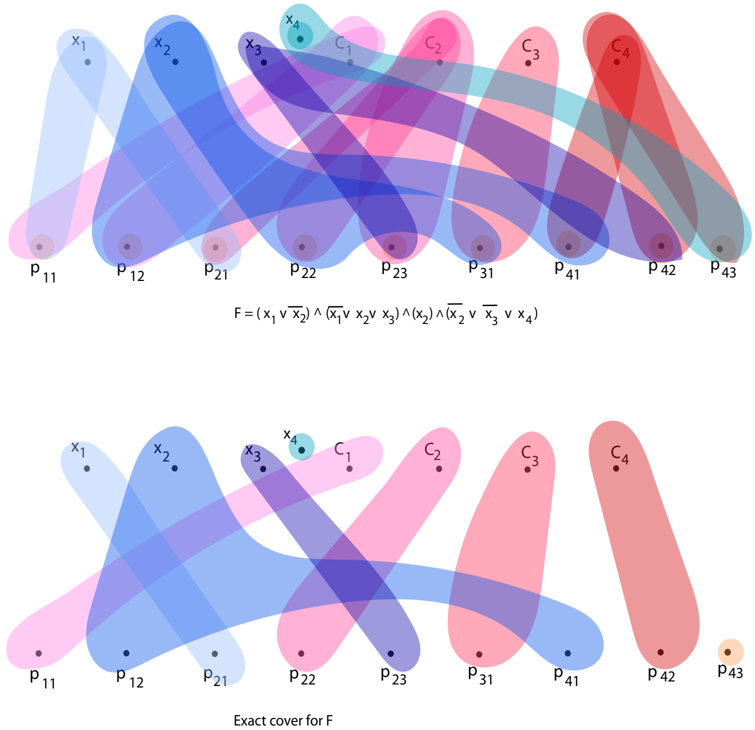


Figure 11.8: Construction of an exact cover from the set of clauses in Example 11.4.

The construction of the graph G uses a trick involving a small subgraph Gad with 7 (distinct) nodes known as a *gadget* shown in Figure 11.9.

The crucial property of the graph Gad is that if Gad is a subgraph of a bigger graph G in such a way that no edge of G is incident to any of the nodes u, v, w unless it is one of the eight edges of Gad incident to the nodes u, v, w, then for any Hamiltonian cycle in G, either the path $(a, u), (u, v), (v, w), (w, b)$ is traversed or the path $(c, w), (w, v), (v, u), (u, d)$ is traversed, but not both.

The reader should convince herself/himself that indeed, any Hamiltonian cycle that does not traverse either the subpath $(a, u), (u, v), (v, w), (w, b)$ from a to b or the subpath $(c, w), (w, v), (v, u), (u, d)$ from c to d will not traverse one of the nodes u, v, w. Also, the fact that node v is traversed exactly once forces only one of the two paths to be traversed but not both. The reader should also convince herself/himself that a smaller graph does not guarantee the desired property.

It is convenient to use the simplified notation with a special type of edge labeled with the exclusive or sign $\oplus$ between the "edges" between a and b and between d and c, as shown in Figure 11.10.

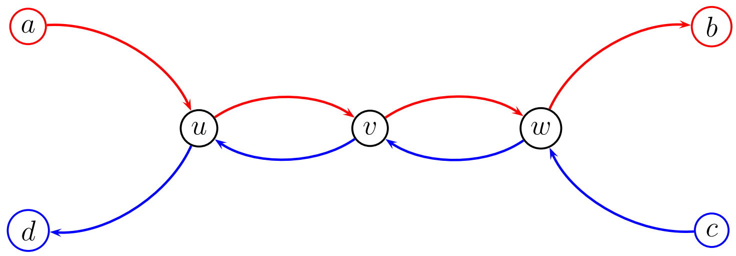


Figure 11.9: A gadget *Gad*.

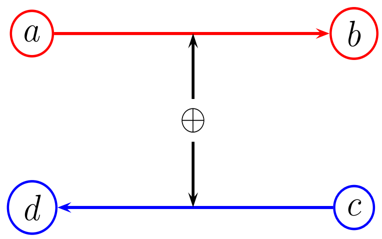


Figure 11.10: A shorthand notation for a gadget.

Whenever such a figure occurs, the actual graph is obtained by substituting a copy of the graph *Gad* (the four nodes a, b, c, d must be distinct). This abbreviating device can be extended to the situation where we build gadgets between a given pair (a, b) and several other pairs $(c_1, d_1), \ldots, (c_m, d_m)$, all nodes being distinct, as illustrated in Figure 11.11.

Either all three edges $(c_1, d_1), (c_2, d_2), (c_3, d_3)$ are traversed or the edge (a, b) is traversed, and these possibilities are mutually exclusive.

The graph $G = \tau(U, \mathcal{F})$ where $U = \{u_1, \ldots, u_n\}$ (with $n \geq 1$) and $\mathcal{F} = \{S_1, \ldots, S_m\}$ (with $m \geq 1$) is constructed as follows:

The graph G has $m + n + 2$ nodes $\{u_0, u_1, \ldots, u_n, S_0, S_1, \ldots, S_m\}$. Note that we have added two extra nodes u_0 and S_0. For $i = 1, \ldots, m$, there are *two* edges $(S_{i-1}, S_i)_1$ and $(S_{i-1}, S_i)_2$ from S_{i-1} to S_i. For $j = 1, \ldots, n$, from u_{j-1} to u_j, there are as many edges as there are sets $S_i \in \mathcal{F}$ containing the element u_j. We can think of each edge between u_{j-1} and u_j as an occurrence of u_j in a uniquely determined set $S_i \in \mathcal{F}$; we denote this edge by $(u_{j-1}, u_j)_i$. We also have an edge from u_n to S_0 and an edge from S_m to u_0, thus "closing the cycle."

What we have constructed so far is not a legal graph since it may have many parallel

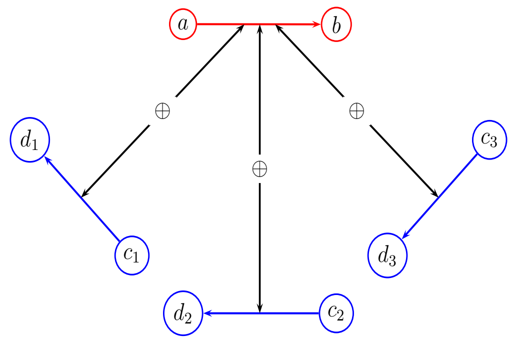


Figure 11.11: A shorthand notation for several gadgets.

edges, but are going to turn it into a legal graph by pairing edges between the u_j's and edges between the S_i's. Indeed, since each edge $(u_{j-1}, u_j)_i$ between u_{j-1} and u_j corresponds to an occurrence of u_j in some uniquely determined set $S_i \in \mathcal{F}$ (that is, $u_j \in S_i$), we put an exclusive-or edge between the edge $(u_{j-1}, u_j)_i$ and the edge $(S_{i-1}, S_i)_2$ between S_{i-1} and S_i, which we call the *long edge*. The other edge $(S_{i-1}, S_i)_1$ between S_{i-1} and S_i (not paired with any other edge) is called the *short edge*. Effectively, we put a copy of the gadget graph *Gad* with $a = u_{j-1}, b = u_j, c = S_{i-1}, d = S_i$ for any pair (u_j, S_i) such that $u_j \in S_i$. The resulting object is indeed a directed graph with no parallel edges. The graph G can be constructed from $(U, \mathcal{F})$ in time $O(n^2)$.

Example 11.5. The above construction is illustrated in Figure 11.12 for the instance of the exact cover problem given by

$$U = \{u_1, u_2, u_3, u_4\},\ \mathcal{F} = \{S_1 = \{u_3, u_4\},\ S_2 = \{u_2, u_3, u_4\},\ S_3 = \{u_1, u_2\}\}.$$

It remains to prove that $(U, \mathcal{F})$ has an exact cover iff the graph $G = \tau(U, \mathcal{F})$ has a Hamiltonian cycle. First, assume that G has a Hamiltonian cycle. If so, for every j some unique "edge" $(u_{j-1}, u_j)_i$ is traversed once (since every u_j is traversed once), and by the exclusive-or nature of the gadget graphs, the corresponding long edge $(S_{i-1}, S_i)_2$ can't be traversed, which means that the short edge $(S_{i-1}, S_i)_1$ is traversed. Consequently, if $\mathcal{C}$ consists of those subsets S_i such that the short edge $(S_{i-1}, S_i)_1$ is traversed, then $\mathcal{C}$ consists of pairwise disjoint subsets whose union is U, namely $\mathcal{C}$ is an exact cover.

In our example, there is a Hamiltonian where the blue edges are traversed between the S_i nodes, and the red edges are traversed between the u_j nodes, namely

$$\begin{aligned}&\text{short } (S_0, S_1),\ \text{long } (S_1, S_2),\ \text{short } (S_2, S_3),\ (S_3, u_0),\\ &(u_0, u_1)_3,\ (u_1, u_2)_3,\ (u_2, u_3)_1,\ (u_3, u_4)_1,\ (u_4, S_0).\end{aligned}$$

The subsets corresponding to the short (S_{i-1}, S_i) edges are S_1 and S_3, and indeed $\mathcal{C} = \{S_1, S_3\}$ is an exact cover.

Note that the exclusive-or property of the gadgets implies the following: since the edge $(u_0, u_1)_3$ must be chosen to obtain a Hamiltonian, the long edge (S_2, S_3) can't be chosen, so the edge $(u_1, u_2)_3$ must be chosen, but then the edge $(u_1, u_2)_2$ is not chosen so the long edge (S_1, S_2) must be chosen, so the edges $(u_2, u_3)_2$ and $(u_3, u_4)_2$ can't be chosen, and thus edges $(u_2, u_3)_1$ and $(u_3, u_4)_1$ must be chosen.

Conversely, if $\mathcal{C}$ is an exact cover for $(U, \mathcal{F})$, then consider the path in G obtained by traversing each short edge $(S_{i-1}, S_i)_1$ for which $S_i \in \mathcal{C}$, each edge $(u_{j-1}, u_j)_i$ such that $u_j \in S_i$, which means that this edge is connected by a $\oplus$-sign to the long edge $(S_{i-1}, S_i)_2$ (by construction, for each u_j there is a unique such S_i), and the edges (u_n, S_0) and (S_m, u_0), then we obtain a Hamiltonian cycle. Observe that the long edges are the inside edges joining the S_i.

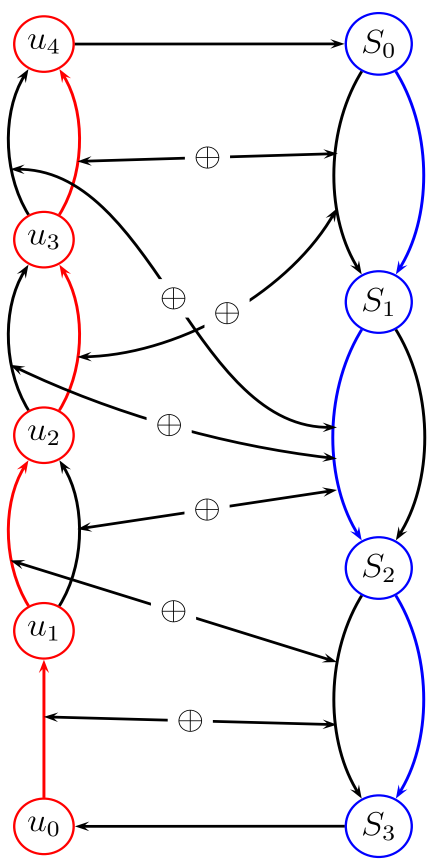


Figure 11.12: The directed graph constructed from the data $(U, \mathcal{F})$ of Example 11.5.

In our example, the exact cover $\mathcal{C} = \{S_1, S_3\}$ yields the Hamiltonian

$$\text{short } (S_0, S_1),\ \text{long } (S_1, S_2),\ \text{short } (S_2, S_3),\ (S_3, u_0),$$
$$(u_0, u_1)_3,\ (u_1, u_2)_3,\ (u_2, u_3)_1,\ (u_3, u_4)_1,\ (u_4, S_0)$$

that we encountered earlier.

(3) **Hamiltonian Cycle (for Undirected Graphs)**

To show that **Hamiltonian Cycle (for Undirected Graphs)** is $\mathcal{NP}$-complete we reduce **Hamiltonian Cycle (for Directed Graphs)** to it:

Hamiltonian Cycle (for Directed Graphs) $\leq_P$ **Hamiltonian Cycle (for Undirected Graphs)**

Given any directed graph $G = (V, E)$ we need to construct in polynomial time an undirected graph $\tau(G) = G' = (V', E')$ such that G has a (directed) Hamiltonian cycle iff G' has a (undirected) Hamiltonian cycle. This is easy. We make three distinct copies v_0, v_1, v_2 of every node $v \in V$ which we put in V', and for every edge $(u, v) \in E$ we create five edges $\{u_0, u_1\}, \{u_1, u_2\}, \{u_2, v_0\}, \{v_0, v_1\}, \{v_1, v_2\}$ which we put in E', as illustrated in the diagram shown in Figure 11.13.

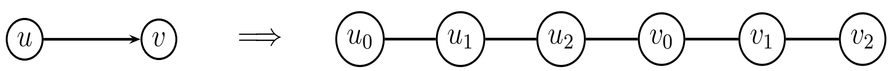


Figure 11.13: Conversion of a directed graph into an undirected graph.

If the size n of the input is $|V| + |E|$, then G' is constructed in time $O(n)$. The crucial point about the graph G' is that although there may be several edges adjacent to a node u_0 or a node u_2, the only way to reach u_1 from u_0 is through the edge $\{u_0, u_1\}$ and the only way to reach u_1 from u_2 is through the edge $\{u_1, u_2\}$.

Suppose there is a Hamiltonian cycle in G'. If this cycle arrives at a node u_0 from the node u_1, then by the above remark, the previous node in the cycle must be u_2. Then the predecessor of u_2 in the cycle must be a node v_0 such that there is an edge $\{u_2, v_0\}$ in G' arising from an edge (u, v) in G. The nodes in the cycle in G' are traversed in the order (v_0, u_2, u_1, u_0) where v_0 and u_2 are traversed in the opposite order in which they occur as the endpoints of the edge (u, v) in G. If so, consider the reverse of our Hamiltonian cycle in G', which is also a Hamiltonian cycle since G' is unoriented. In this cycle, we go from u_0 to u_1, then to u_2, and finally to v_0. In G, we traverse the edge from u to v. In order for the cycle in G' to be Hamiltonian, we must continue by visiting v_1 and v_2, since otherwise v_1 is never traversed. Now the next node w_0 in the Hamiltonian cycle in G' corresponds to an edge (v, w) in G, and by repeating our reasoning we see that our Hamiltonian cycle in G' determines a Hamiltonian cycle in G. We leave it as an easy exercise to check that a Hamiltonian cycle in G yields a Hamiltonian cycle in G'. The process of expanding a directed graph into an undirected graph and the inverse process are illustrated in Figure 11.14 and Figure 11.15.

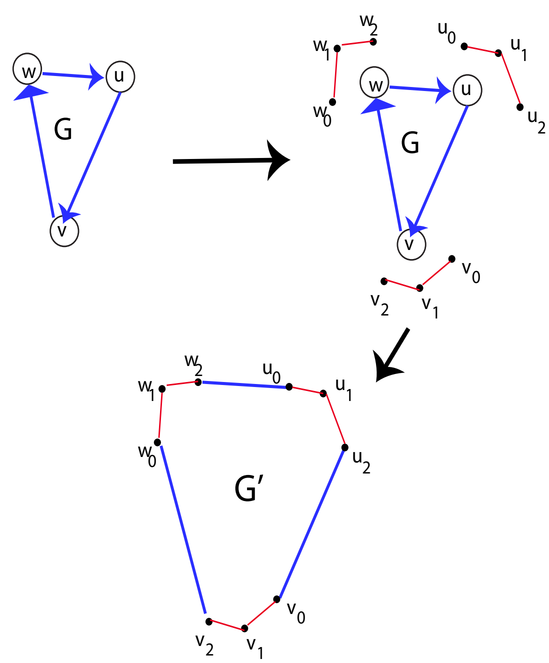


Figure 11.14: Expanding the directed graph into an undirected graph.

(4) **Traveling Salesman Problem**

To show that the **Traveling Salesman Problem** is $\mathcal{NP}$-complete, we reduce the **Hamiltonian Cycle Problem (Undirected Graphs)** to it:

Hamiltonian Cycle Problem (Undirected Graphs) $\leq_P$ **Traveling Salesman Problem**

This is a fairly easy reduction.

Given an undirected graph $G = (V, E)$, we construct an instance $\tau(G) = (D, B)$ of the Traveling Salesman Problem so that G has a Hamiltonian cycle iff the traveling salesman problem has a solution. If we let $n = |V|$, we have n cities and the matrix

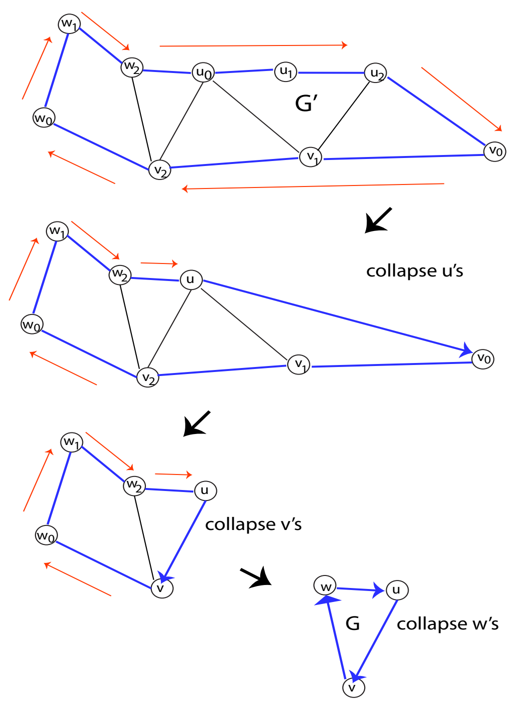


Figure 11.15: Collapsing the undirected graph onto a directed graph.

$D = (d_{ij})$ is defined as follows:

$$d_{ij} = \begin{cases} 0 & \text{if } i = j \\ 1 & \text{if } \{v_i, v_j\} \in E \\ 2 & \text{otherwise.} \end{cases}$$

We also set the budget B as $B = n$. The construction of (D, B) from G can be done in time $O(n^2)$.

Any tour of the cities has cost equal to n plus the number of pairs (v_i, v_j) such that $i \neq j$ and $\{v_i, v_j\}$ is *not* an edge of G. It follows that a tour of cost n exists iff there are no pairs (v_i, v_j) of the second kind iff the tour is a Hamiltonian cycle.

The reduction from **Hamiltonian Cycle Problem (Undirected Graphs)** to the **Traveling Salesman Problem** is quite simple, but a direct reduction of say **Satisfiability** to the **Traveling Salesman Problem** is hard. By breaking this reduction into several steps made it simpler to achieve.

(5) **Independent Set**

To show that **Independent Set** is $\mathcal{NP}$-complete, we reduce **Exact 3-Satisfiability** to it:

Exact 3-Satisfiability $\leq_P$ Independent Set

Recall that in **Exact 3-Satisfiability** every clause C_i has exactly three literals L_{i1}, L_{i2}, L_{i3}.

Given a set $F = \{C_1, \ldots, C_m\}$ of $m \geq 2$ such clauses, we construct in polynomial time an undirected graph $G = (V, E)$ such that F is satisfiable iff G has an independent set C with at least $K = m$ nodes.

For every i ($1 \leq i \leq m$), we have three nodes c_{i1}, c_{i2}, c_{i3} corresponding to the three literals L_{i1}, L_{i2}, L_{i3} in clause C_i, so there are $3m$ nodes in V. The "core" of G consists of m triangles, one for each set $\{c_{i1}, c_{i2}, c_{i3}\}$. We also have an edge $\{c_{ik}, c_{j\ell}\}$ iff L_{ik} and $L_{j\ell}$ are complementary literals. If the size n of the input is the sum of the lengths of the clauses, then the construction of G can be done in time $O(n^2)$.

Example 11.6. Let F be the set of clauses

$$F = \{C_1 = (x_1 \vee \overline{x_2} \vee x_3),\ C_2 = (\overline{x_1} \vee \overline{x_2} \vee x_3),\ C_3 = (\overline{x_1} \vee \overline{x_2} \vee \overline{x_3}),\ C_4 = (x_1 \vee x_2 \vee x_3)\}.$$

The graph G associated with F is shown in Figure 11.16.

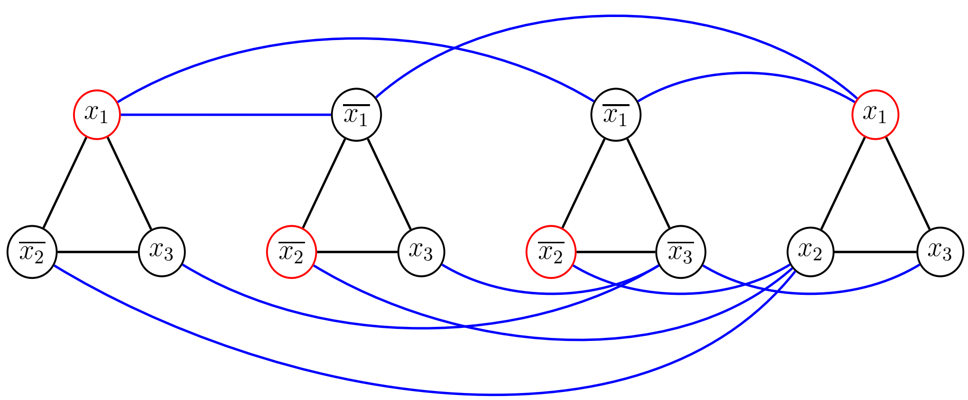


Figure 11.16: The graph constructed from the clauses of Example 11.6.

It remains to show that the construction works. Since any three nodes in a triangle are connected, an independent set C can have at most one node per triangle and thus has at most m nodes. Since the budget is $K = m$, we may assume that there is an independent set with m nodes. Define a (partial) truth assignment by

$$v(x_i) = \begin{cases} \mathbf{T} & \text{if } L_{jk} = x_i \text{ and } c_{jk} \in C \\ \mathbf{F} & \text{if } L_{jk} = \overline{x_i} \text{ and } c_{jk} \in C. \end{cases}$$

Since the non-triangle edges in G link nodes corresponding to complementary literals and nodes in C are not connected, our truth assigment does not assign clashing truth values to the variables x_i. Not all variables may receive a truth value, in which case we assign an arbitrary truth value to the unassigned variables. This yields a satisfying assignment for F.

In Example 11.6, the set $C = \{c_{11}, c_{22}, c_{32}, c_{41}\}$ corresponding to the nodes shown in red in Figure 11.16 form an independent set, and they induce the partial truth assignment $v(x_1) = \mathbf{T}, v(x_2) = \mathbf{F}$. The variable x_3 can be assigned an arbitrary value, say $v(x_3) = \mathbf{F}$, and v is indeed a satisfying truth assignment for F.

Conversely, if v is a truth assignment for F, then we obtain an independent set C of size m by picking for each clause C_i a node c_{ik} corresponding to a literal L_{ik} whose value under v is $\mathbf{T}$.

(6) **Clique**

To show that **Clique** is $\mathcal{NP}$-complete, we reduce **Independent Set** to it:

Independent Set $\leq_P$ **Clique**

The key to the reduction is the notion of the complement of an undirected graph $G = (V, E)$. The *complement* $G^c = (V, E^c)$ of the graph $G = (V, E)$ is the graph with the same set of nodes V as G but there is an edge $\{u, v\}$ (with $u \neq v$) in E^c iff $\{u, v\} \notin E$. Then it is not hard to check that there is a bijection between maximum independent sets in G and maximum cliques in G^c. The reduction consists in constructing from a graph G its complement G^c, and then G has an independent set iff G^c has a clique. Obviously, the reduction can be done in linear time.

This construction is illustrated in Figure 11.17, where a maximum independent set in the graph G is shown in blue and a maximum clique in the graph G^c is shown in red.

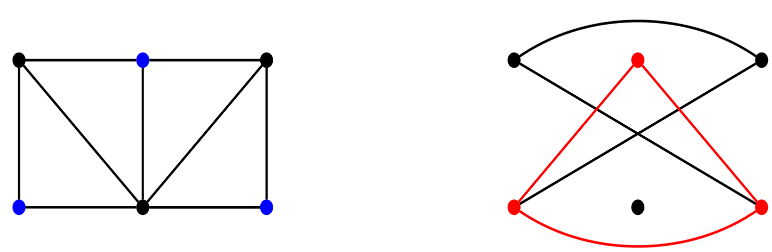


Figure 11.17: A graph (left) and its complement (right).

(7) **Node Cover**

To show that **Node Cover** is $\mathcal{NP}$-complete, we reduce **Independent Set** to it:

Independent Set $\leq_P$ Node Cover

This time the crucial observation is that if N is an independent set in G, then the complement $C = V - N$ of N in V is a node cover in G. Thus there is an independent set of size at least K iff there is a node cover of size at most $n - K$ where $n = |V|$ is the number of nodes in V. The reduction leaves the graph unchanged and replaces K by $n - K$. Obviously, the reduction can be done in linear time. An example is shown in Figure 11.18 where an independent set is shown in blue and a node cover is shown in red.

Figure 11.18: An inpendent set (left) and a node cover (right).

(8) **Knapsack (also called Subset sum)**

To show that **Knapsack** is $\mathcal{NP}$-complete, we reduce **Exact Cover** to it:

Exact Cover $\leq_P$ Knapsack

Given an instance $(U, \mathcal{F})$ of set cover with $U = \{u_1, \ldots, u_n\}$ and $\mathcal{F} = \{S_1, \ldots, S_m\}$, a family of subsets of U, we need to produce in polynomial time an instance $\tau(U, \mathcal{F})$ of the Knapsack Problem consisting of k nonnegative integers $a_1, \ldots, a_k$ and another integer $K > 0$ such that there is a subset $I \subseteq \{1, \ldots, k\}$ such that $\sum_{i \in I} a_i = K$ iff there is an exact cover of U using subsets in $\mathcal{F}$.

The trick here is the relationship between *set union* and *integer addition*.

Example 11.7. Consider the exact cover problem given by $U = \{u_1, u_2, u_3, u_4\}$ and

$$\mathcal{F} = \{S_1 = \{u_3, u_4\},\ S_2 = \{u_2, u_3, u_4\},\ S_3 = \{u_1, u_2\}\}.$$

We can represent each subset S_j by a binary string a_j of length 4, where the ith bit from the left is 1 iff $u_i \in S_j$, and 0 otherwise. In our example

$$\begin{aligned}a_1 &= 0011\\ a_2 &= 0111\\ a_3 &= 1100.\end{aligned}$$

Then the trick is that some family $\mathcal{C}$ of subsets S_j is an exact cover if the sum of the corresponding numbers a_j adds up to $1111 = 2^4 - 1 = K$. For example,

$$\mathcal{C} = \{S_1 = \{u_3, u_4\},\ S_3 = \{u_1, u_2\}\}$$

is an exact cover and

$$a_1 + a_3 = 0011 + 1100 = 1111.$$

Unfortunately, there is a problem with this encoding which has to do with the fact that addition may involve carry. For example, assuming four subsets and the universe $U = \{u_1, \ldots, u_6\}$,

$$11 + 13 + 15 + 24 = 63,$$

in binary

$$001011 + 001101 + 001111 + 011000 = 111111,$$

but if we convert these binary strings to the corresponding subsets we get the subsets

$$\begin{aligned}S_1 &= \{u_3, u_5, u_6\}\\ S_2 &= \{u_3, u_4, u_6\}\\ S_3 &= \{u_3, u_4, u_5, u_6\}\\ S_4 &= \{u_2, u_3\},\end{aligned}$$

which are not disjoint and do not cover U.

The fix is surprisingly simple: use base m (where m is the number of subsets in $\mathcal{F}$) instead of base 2.

Example 11.8. Consider the exact cover problem given by $U = \{u_1, u_2, u_3, u_4, u_5, u_6\}$ and $\mathcal{F}$ given by

$$\begin{aligned}S_1 &= \{u_3, u_5, u_6\}\\ S_2 &= \{u_3, u_4, u_6\}\\ S_3 &= \{u_3, u_4, u_5, u_6\}\\ S_4 &= \{u_2, u_3\},\\ S_5 &= \{u_1, u_2, u_4\}.\end{aligned}$$

In base $m = 5$, the numbers corresponding to $S_1, \ldots, S_5$ are

$$\begin{aligned}a_1 &= 001011\\ a_2 &= 001101\\ a_3 &= 001111\\ a_4 &= 011000\\ a_5 &= 110100.\end{aligned}$$

This time,

$$a_1 + a_2 + a_3 + a_4 = 001011 + 001101 + 001111 + 011000 = 014223 \neq 111111,$$

so $\{S_1, S_2, S_3, S_4\}$ is not a solution. However

$$a_1 + a_5 = 001011 + 110100 = 111111,$$

and $\mathcal{C} = \{S_1, S_5\}$ is an exact cover.

Thus, given an instance $(U, \mathcal{F})$ of **Exact Cover** where $U = \{u_1, \ldots, u_n\}$ and $\mathcal{F} = \{S_1, \ldots, S_m\}$ the reduction to **Knapsack** consists in forming the m numbers $a_1, \ldots, a_m$ (each of n bits) encoding the subsets S_j, namely $a_{ji} = 1$ iff $u_i \in S_j$, else 0, and to let $K = 1 + m^2 + \cdots + m^{n-1}$, which is represented in base m by the string $\underbrace{11 \cdots 11}_{n}$. In testing whether $\sum_{i \in I} a_i = K$ for some subset $I \subseteq \{1, \ldots, m\}$, we use arithmetic in base m.

If a candidate solution $\mathcal{C}$ involves at most $m - 1$ subsets, then since the corresponding numbers are added in base m, a carry can never happen. If the candidate solution involves all m subsets, then $a_1 + \cdots + a_m = K$ iff $\mathcal{F}$ is a partition of U, since otherwise some bit in the result of adding up these m numbers in base m is not equal to 1, even if a carry occurs. Since the number K is written in binary, it takes time $O(mn)$ to produce $((a_1, \ldots, a_m), K)$ from $(U, \mathcal{F})$.

(9) **Inequivalence of $*$-free Regular Expressions**

To show that **Inequivalence of $*$-free Regular Expressions** is $\mathcal{NP}$-complete, we reduce the **Satisfiability Problem** to it:

Satisfiability Problem $\leq_P$ Inequivalence of $*$-free Regular Expressions

We already argued that **Inequivalence of $*$-free Regular Expressions** is in $\mathcal{NP}$ because if R is a $*$-free regular expression, then for every string $w \in \mathcal{L}[R]$ we have $|w| \leq |R|$. The above observation shows that if R_1 and R_2 are $*$-free and if there is a string $w \in (\mathcal{L}[R_1] - \mathcal{L}[R_2]) \cup (\mathcal{L}[R_2] - \mathcal{L}[R_1])$, then $|w| \leq |R_1| + |R_2|$, so we can indeed check this in polynomial time. It follows that the inequivalence problem for $*$ -free regular expressions is in $\mathcal{NP}$.

We reduce the **Satisfiability Problem** to the **Inequivalence of $*$-free Regular Expressions** as follows. For any set of clauses $P = C_1 \wedge \cdots \wedge C_p$, if the propositional variables occurring in P are $x_1, \ldots, x_n$, we produce two $*$-free regular expressions R, S over $\Sigma = \{0, 1\}$, such that P is satisfiable iff $L_R \neq L_S$. The expression S is actually

$$S = \underbrace{(0+1)(0+1)\cdots(0+1)}_{n}.$$

The expression R is of the form

$$R = R_1 + \cdots + R_p,$$

where R_i is constructed from the clause C_i in such a way that L_{R_i} corresponds precisely to the set of truth assignments that falsify C_i; see below.

Given any clause C_i, let R_i be the $*$-free regular expression defined such that, if x_j and $\overline{x}_j$ both belong to C_i (for some j), then $R_i = \emptyset$, else

$$R_i = R_i^1 \cdot R_i^2 \cdots R_i^n,$$

where R_i^j is defined by

$$R_i^j = \begin{cases} 0 & \text{if } x_j \text{ is a literal of } C_i \\ 1 & \text{if } \overline{x}_j \text{ is a literal of } C_i \\ (0+1) & \text{if } x_j \text{ does not occur in } C_i. \end{cases}$$

The construction of R from P takes linear time.

Example 11.9. If we apply the above conversion to the clauses of Example 11.3, namely

$$F = \{C_1 = (x_1 \vee \overline{x_2}),\ C_2 = (\overline{x_1} \vee x_2 \vee x_3),\ C_3 = (x_2),\ C_4 = (\overline{x_2} \vee \overline{x_3})\},$$

we get

$$R_1 = 0 \cdot 1 \cdot (0+1), \quad R_2 = 1 \cdot 0 \cdot 0, \quad R_3 = (0+1) \cdot 0 \cdot (0+1), \quad R_4 = (0+1) \cdot 1 \cdot 1.$$

Clearly, all truth assignments that falsify C_i must assign $\mathbf{F}$ to x_j if $x_j \in C_i$ or assign $\mathbf{T}$ to x_j if $\overline{x}_j \in C_i$. Therefore, L_{R_i} corresponds to the set of truth assignments that falsify C_i (where 1 stands for $\mathbf{T}$ and 0 stands for $\mathbf{F}$) and thus, if we let

$$R = R_1 + \cdots + R_p,$$

then L_R corresponds to the set of truth assignments that falsify $P = C_1 \wedge \cdots \wedge C_p$. Since $L_S = \{0,1\}^n$ (all binary strings of length n), we conclude that $L_R \neq L_S$ iff P is satisfiable. Therefore, we have reduced the **Satisfiability Problem** to our problem and the reduction clearly runs in polynomial time. This proves that the problem of deciding whether $L_R \neq L_S$, for any two $*$-free regular expressions R and S is $\mathcal{NP}$-complete.

(10) 0-1 **integer programming problem**

It is easy to check that the problem is in $\mathcal{NP}$.

To prove that the is $\mathcal{NP}$-complete we reduce the **bounded-tiling problem** to it:

bounded-tiling problem $\leq_P$ 0-1 **integer programming problem**

Given a tiling problem, $((\mathcal{T}, V, H), \widehat{s}, \sigma_0)$, we create a 0-1-valued variable x_{mnt}, such that $x_{mnt} = 1$ iff tile t occurs in position (m, n) in some tiling. Write equations or inequalities expressing that a tiling exists and then use "slack variables" to convert inequalities to equations. For example, to express the fact that every position is tiled by a single tile, use the equation

$$\sum_{t \in \mathcal{T}} x_{mnt} = 1,$$

for all m, n with $1 \leq m \leq 2s$ and $1 \leq n \leq s$. We leave the rest as as exercise.

11.3 Succinct Certificates, co$\mathcal{NP}$, and $\mathcal{EXP}$

All the problems considered in Section 11.1 share a common feature, which is that for each problem, a solution is produced nondeterministically (an exact cover, a directed Hamiltonian cycle, a tour of cities, an independent set, a node cover, a clique *etc.*), and then this candidate solution is checked deterministically and in polynomial time. The candidate solution is a string called a *certificate* (or *witness*).

It turns out that membership on $\mathcal{NP}$ can be defined in terms of certificates. To be a certificate, a string must satisfy two conditions:

1. It must be *polynomially succinct*, which means that its length is at most a polynomial in the length of the input.

2. It must be *checkable* in polynomial time.

All "yes" inputs to a problem in $\mathcal{NP}$ must have at least one certificate, while all "no" inputs must have none.

The notion of certificate can be formalized using the notion of a polynomially balanced language.

Definition 11.3. Let Σ be an alphabet, and let ";" be a symbol not in Σ. A language $L' \subseteq \Sigma^*; \Sigma^*$ is said to be *polynomially balanced* if there exists a polynomial $p(X)$ such that for all $x, y \in \Sigma^*$, if $x; y \in L'$ then $|y| \leq p(|x|)$.

Suppose L' is a polynomially balanced language and that $L' \in \mathcal{P}$. Then we can consider the language

$$L = \{x \in \Sigma^* \mid (\exists y \in \Sigma^*)(x; y \in L')\}.$$

The intuition is that for each $x \in L$, the set

$$\{y \in \Sigma^* \mid x; y \in L'\}$$

is the set of certificates of x. For every $x \in L$, a Turing machine can nondeterministically guess one of its certificates y, and then use the deterministic Turing machine for L' to check in polynomial time that $x; y \in L'$. Note that, by definition, strings not in L have no certificate. It follows that $L \in \mathcal{NP}$.

Conversely, if $L \in \mathcal{NP}$ and the alphabet Σ has at least two symbols, we can encode the paths in the computation tree for every input $x \in L$, and we obtain a polynomially balanced language $L' \subseteq \Sigma^*; \Sigma^*$ with L' in $\mathcal{P}$ such that

$$L = \{x \in \Sigma^* \mid (\exists y \in \Sigma^*)(x; y \in L')\}.$$

The details of this construction are left as an exercise. In summary, we obtain the following theorem.

Theorem 11.1. *Let $L \subseteq \Sigma^*$ be a language over an alphabet Σ with at least two symbols, and let ";" be a symbol not in Σ. Then $L \in \mathcal{NP}$ iff there is a polynomially balanced language $L' \subseteq \Sigma^*; \Sigma^*$ such that $L' \in \mathcal{P}$ and*

$$L = \{x \in \Sigma^* \mid (\exists y \in \Sigma^*)(x; y \in L')\}.$$

A striking illustration of the notion of succint certificate is illustrated by the set of *composite* integers, namely those natural numbers $n \in \mathbb{N}$ that can be written as the product pq of two numbers $p, q \geq 2$ with $p, q \in \mathbb{N}$. For example, the number

$$4,294,967,297$$

is a composite!

This is far from obvious, but if an oracle gives us the certificate $\{6,700,417,\ \ 641\}$, it is easy to carry out in polynomial time the multiplication of these two numbers and check that it is equal to $4,294,967,297$. Finding a certificate is usually (very) hard, but checking that it works is easy. This is the point of certificates.

We conclude this section with a brief discussion of the complexity classes co$\mathcal{NP}$ and $\mathcal{EXP}$.

By definition,

$$\mathrm{co}\mathcal{NP} = \{\overline{L} \mid L \in \mathcal{NP}\},$$

that is, co$\mathcal{NP}$ consists of all complements of languages in $\mathcal{NP}$. Since $\mathcal{P} \subseteq \mathcal{NP}$ and $\mathcal{P}$ is closed under complementation,

$$\mathcal{P} \subseteq \mathrm{co}\mathcal{NP},$$

but nobody knows whether $\mathcal{NP}$ is closed under complementation, that is, nobody knows whether $\mathcal{NP} = \mathrm{co}\mathcal{NP}$.

A language L is *co$\mathcal{NP}$-hard* if every language in co$\mathcal{NP}$ is polynomial-time reducible to L, and *co$\mathcal{NP}$-complete* if $L \in \mathrm{co}\mathcal{NP}$ and L is co$\mathcal{NP}$-hard.

What can be shown is that if $\mathcal{NP} \neq \mathrm{co}\mathcal{NP}$, then $\mathcal{P} \neq \mathcal{NP}$. However it is possible that $\mathcal{P} \neq \mathcal{NP}$ and yet $\mathcal{NP} = \mathrm{co}\mathcal{NP}$, although this is considered unlikely.

Of course, $\mathcal{P} \subseteq \mathcal{NP} \cap \mathrm{co}\mathcal{NP}$. There are problems in $\mathcal{NP} \cap \mathrm{co}\mathcal{NP}$ not known to be in $\mathcal{P}$. One of the most famous in the following problem:

Integer factorization problem:

Given an integer $N \geq 3$, and another integer M (a budget) such that $1 < M < N$, does N have a factor d with $1 < d \leq M$?

Proposition 11.2. *The problem* **Integer factorization** *is in $\mathcal{NP} \cap \mathrm{co}\mathcal{NP}$.*

Proof. That **Integer factorization** is in $\mathcal{NP}$ is clear. To show that **Integer factorization** is in co$\mathcal{NP}$, we can guess a factorization of N into distinct factors all greater than M, check that they are prime using the results of Chapter 12 showing that testing primality is in $\mathcal{NP}$ (even in $\mathcal{P}$, but that's much harder to prove), and then check that the product of these factors is N. $\square$

It is widely believed that **Integer factorization** *does not* belong to $\mathcal{P}$, which is the technical justification for saying that this problem is hard. Most cryptographic algorithms rely on this unproven fact. If **Integer factorization** was either $\mathcal{NP}$-complete or co$\mathcal{NP}$-complete, then we would have $\mathcal{NP} = \mathrm{co}\mathcal{NP}$, which is considered very unlikely.

Remark: If $\sqrt{N} \leq M < N$, the above problem is equivalent to asking whether N is prime.

A natural instance of a problem in co$\mathcal{NP}$ is the *unsatisfiability problem* for propositions UNSAT $= \neg$SAT, namely deciding that a proposition P has no satisfying assignment.

Definition 11.4. A proposition P (in CNF) is *falsifiable* if there is some truth assigment v such that $\widehat{v}(P) = \mathbf{F}$.

It is obvious that the set of falsifiable propositions is in $\mathcal{NP}$. Since a proposition P is valid iff P is not falsifiable, the *validity (or tautology) problem* TAUT for propositions is in co$\mathcal{NP}$. In fact, the follolwing result holds.

Proposition 11.3. *The problem* TAUT *is* co$\mathcal{NP}$*-complete.*

Proof. See Papadimitriou [38]. Since SAT is $\mathcal{NP}$-complete, for every language $L \in \mathcal{NP}$, there is a polynomial-time computable function $f\colon \Sigma^* \to \Sigma^*$ such that $x \in L$ iff $f(x) \in \mathrm{SAT}$. Then $x \notin L$ iff $f(x) \notin \mathrm{SAT}$, that is, $x \in \overline{L}$ iff $f(x) \in \neg\mathrm{SAT}$, which means that every language $\overline{L} \in \mathrm{co}\mathcal{NP}$ is polynomial-time reducible to $\neg\mathrm{SAT} = \mathrm{UNSAT}$. But $\mathrm{TAUT} = \{\neg P \mid P \in \mathrm{UNSAT}\}$, so we have the polynomial-time computable function g given by $g(x) = \neg f(x)$ which gives us the reduction $x \in \overline{L}$ iff $g(x) \in \mathrm{TAUT}$, which shows that TAUT is co$\mathcal{NP}$-complete. ☐

Despite the fact that this problem has been extensively studied, not much is known about its exact complexity.

The reasoning used to show that TAUT is co$\mathcal{NP}$-complete can also be used to show the following interesting result.

Proposition 11.4. *If a language L is $\mathcal{NP}$-complete, then its complement $\overline{L}$ is* co$\mathcal{NP}$*-complete.*

Proof. By definition, since $L \in \mathcal{NP}$, we have $\overline{L} \in \mathrm{co}\mathcal{NP}$. Since L is $\mathcal{NP}$-complete, for every language $L_2 \in \mathcal{NP}$, there is a polynomial-time computable function $f\colon \Sigma^* \to \Sigma^*$ such that $x \in L_2$ iff $f(x) \in L$. Then $x \notin L_2$ iff $f(x) \notin L$, that is, $x \in \overline{L_2}$ iff $f(x) \in \overline{L}$, which means that $\overline{L}$ is co$\mathcal{NP}$-hard as well, thus co$\mathcal{NP}$-complete. ☐

The class $\mathcal{EXP}$ is defined as follows.

Definition 11.5. A deterministic Turing machine M is said to be *exponentially bounded* if there is a polynomial $p(X)$ such that for every input $x \in \Sigma^*$, there is no ID ID_n such that

$$ID_0 \vdash ID_1 \vdash^* ID_{n-1} \vdash ID_n, \quad \text{with} \quad n > 2^{p(|x|)}.$$

The class $\mathcal{EXP}$ is the class of all languages that are accepted by some exponentially bounded deterministic Turing machine.

Remark: We can also define the class $\mathcal{NEXP}$ as in Definition 11.5, except that we allow nondeterministic Turing machines.

One of the interesting features of $\mathcal{EXP}$ is that it contains $\mathcal{NP}$.

Theorem 11.5. *We have the inclusion $\mathcal{NP} \subseteq \mathcal{EXP}$.*

Sketch of proof. Let M be some nondeterministic Turing machine accepting L in polynomial time bounded by $p(X)$. We can construct a deterministic Turing machine M' that operates as follows: for every input x, M' simulates M on all computations of length 1, then on all possible computations of length 2, and so on, up to all possible computations of length

$p(|x|)+1$. At this point, either an accepting computation has been discovered or all computations have halted rejecting. We claim that M' operates in time bounded by $2^{q(|x|)}$ for some polynomial $q(X)$. First, let r be the degree of nondeterminism of M, that is, the maximum number of triples (b,m,q) such that a quintuple (p,q,b,m,q) is an instructions of M. Then to simulate a computation of M of length ℓ, M' needs $O(\ell)$ steps—to copy the input, to produce a string c in $\{1,\ldots,r\}^\ell$, and so simulate M according to the choices specified by c. It follows that M' can carry out the simulation of M on an input x in

$$\sum_{\ell=1}^{p(|x|)+1} r^\ell \leq (r+1)^{p(|x|)+1}$$

steps. Including the $O(\ell)$ extra steps for each ℓ, we obtain the bound $(r+2)^{p(|x|)+1}$. Then we can pick a constant k such that $2^k > r+2$, and with $q(X) = k(p(X)+1)$, we see that M' operates in time bounded by $2^{q(|x|)}$. ☐

It is also immediate to see that $\mathcal{EXP}$ is closed under complementation. Furthermore the strict inclusion $\mathcal{P} \subset \mathcal{EXP}$ holds.

Theorem 11.6. *We have the strict inclusion $\mathcal{P} \subset \mathcal{EXP}$.*

Sketch of proof. We use a diagonalization argument to produce a language E such that $E \notin \mathcal{P}$, yet $E \in \mathcal{EXP}$. We need to code a Turing machine as a string, but this can certainly be done using the techniques of Chapter 4. Let $\#(M)$ be the code of Turing machine M and let $\#(x)$ be the code of x. Define E as

$$E = \{\#(M)\#(x) \mid M \text{ accepts input } x \text{ after at most } 2^{|x|} \text{ steps}\}.$$

We claim that $E \notin \mathcal{P}$. We proceed by contradiction. If $E \in \mathcal{P}$, then so is the language E_1 given by

$$E_1 = \{\#(M) \mid M \text{ accepts } \#(M) \text{ after at most } 2^{|\#(M)|} \text{ steps}\}.$$

Since $\mathcal{P}$ is closed under complementation, we also have $\overline{E_1} \in \mathcal{P}$. Let M^* be a deterministic Turing machine accepting $\overline{E_1}$ in time $p(X)$, for some polynomial $p(X)$. Since $p(X)$ is a polynomial, there is some n_0 such that $p(n) \leq 2^n$ for all all $n \geq n_0$. We may also assume that $|\#(M^*)| \geq n_0$, since if not we can add n_0 "dead states" to M^*.

Now what happens if we run M^* on its own code $\#(M^*)$?

It is easy to see that we get a contradiction, namely M^* accepts $\#(M^*)$ iff M^* rejects $\#(M^*)$. We leave this verification as an exercise.

In conclusion, $\overline{E_1} \notin \mathcal{P}$, which in turn implies that $E \notin \mathcal{P}$.

It remains to prove that $E \in \mathcal{EXP}$. This is because we can construct a Turing machine that can in exponential time simulate any Turing machine M on input x for $2^{|x|}$ steps. ☐

In summary, we have the chain of inclusions

$$\mathcal{P} \subseteq \mathcal{NP} \subseteq \mathcal{EXP},$$

where the inclusions $\mathcal{P} \subset \mathcal{EXP}$ is strict but the left inclusion and the right inclusion are both open problems, and we know that at least one of these two inclusions is strict.

We also have the inclusions

$$\mathcal{P} \subseteq \mathcal{NP} \subseteq \mathcal{EXP} \subseteq \mathcal{NEXP},$$

where the inclusions $\mathcal{P} \subset \mathcal{EXP}$ and $\mathcal{NP} \subset \mathcal{NEXP}$ are strict. The left inclusion and the right inclusion in $\mathcal{NP} \subseteq \mathcal{EXP} \subseteq \mathcal{NEXP}$ are both open problems, but we know that at least one of these two inclusions is strict. It can be shown that if $\mathcal{EXP} \neq \mathcal{NEXP}$, then $\mathcal{P} \neq \mathcal{NP}$; see Papadimitriou [38].

Chapter 12

Primality Testing is in $\mathcal{NP}$

12.1 Prime Numbers and Composite Numbers

Prime numbers have fascinated mathematicians and more generally curious minds for thousands of years. What is a prime number? Well, $2, 3, 5, 7, 11, 13, \ldots, 9973$ are prime numbers.

Definition 12.1. A positive integer p is *prime* if $p \geq 2$ and if p is only divisible by 1 and p. Equivalently, p is prime if and only if p is a positive integer $p \geq 2$ that is not divisible by any integer m such that $2 \leq m < p$. A positive integer $n \geq 2$ which is not prime is called *composite*.

Observe that the number 1 is considered neither a prime nor a composite. For example, $6 = 2 \cdot 3$ is composite. Is $3\,215\,031\,751$ composite? Yes, because

$$3\,215\,031\,751 = 151 \cdot 751 \cdot 28351.$$

Even though the definition of primality is very simple, the structure of the set of prime numbers is highly nontrivial. The prime numbers are the basic building blocks of the natural numbers because of the following theorem bearing the impressive name of *fundamental theorem of arithmetic*.

Theorem 12.1. *Every natural number $n \geq 2$ has a unique factorization*

$$n = p_1^{i_1} p_2^{i_2} \cdots p_k^{i_k},$$

where the exponents $i_1, \ldots, i_k$ are positive integers and $p_1 < p_2 < \cdots < p_k$ are primes.

Every book on number theory has a proof of Theorem 12.1. The proof is not difficult and uses induction. It has two parts. The first part shows the existence of a factorization. The second part shows its uniqueness. For example, see Apostol [2] (Chapter 1, Theorem 1.10).

How many prime numbers are there? Many! In fact, infinitely many.

Theorem 12.2. *The set of prime numbers is infinite.*

Proof. The following proof attributed to Hermite only use the fact that every integer greater than 1 has some prime divisor. We prove that for every natural number $n \geq 2$, there is some prime $p > n$. Consider $N = n! + 1$. The number N must be divisible by some prime p ($p = N$ is possible). Any prime p dividing N is distinct from $2, 3, \ldots, n$, since otherwise p would divide $N - n! = 1$, a contradiction. $\square$

The problem of determining whether a given integer is prime is one of the better known and most easily understood problems of pure mathematics. This problem has caught the interest of mathematicians again and again for centuries. However, it was not until the 20th century that questions about primality testing and factoring were recognized as problems of practical importance and a central part of applied mathematics. The advent of cryptographic systems that use large primes, such as RSA, was the main driving force for the development of fast and reliable methods for primality testing. Indeed, in order to create RSA keys, one needs to produce large prime numbers.

12.2 Methods for Primality Testing

The general strategy to test whether an integer $n > 2$ is prime or composite is to choose some property, say A, implied by primality, and to search for a counterexample a to this property for the number n, namely some a for which property A fails. We look for properties for which checking that a candidate a is indeed a countexample can be done quickly.

A simple property that is the basis of several primality testing algorithms is the *Fermat test*, namely

$$a^{n-1} \equiv 1 \pmod{n},$$

which means that $a^{n-1} - 1$ is divisible by n (see Definition 12.2 for the meaning of the notation $a \equiv b \pmod{n}$). If n is prime, and if $\gcd(a, n) = 1$, then the above test is indeed satisfied; this is Fermat's little theorem, Theorem 12.7.

Typically, together with the number n being tested for primality, some candidate counterexample a is supplied to an algorithm which runs a test to determine whether a is really a counterexample to property A for n. If the test says that a is a counterexample, also called a *witness*, then we know for sure that n is composite.

For example, using the Fermat test, if $n = 10$ and $a = 3$, we check that

$$3^9 = 19683 = 10 \cdot 1968 + 3,$$

so $3^9 - 1$ is not divisible by 10, which means that

$$a^{n-1} = 3^9 \not\equiv 1 \pmod{10},$$

and the Fermat test fails. This shows that 10 is not prime and that $a = 3$ is a witness of this fact.

If the algorithm reports that a is not a witness to the fact that n is composite, does this imply that n is prime? Unfortunately, no. This is because, there may be some composite number n and some candidate counterexample a for which the test says that a is not a counterexample. Such a number a is called a *liar*.

For example, using the Fermat test for $n = 91 = 7 \cdot 13$ and $a = 3$, we can check that

$$a^{n-1} = 3^{90} \equiv 1 \pmod{91},$$

so the Fermat test succeeds even though 91 is not prime. The number $a = 3$ is a liar.

The other reason is that we haven't tested all the candidate counterexamples a for n. In the case where $n = 91$, it can be shown that $2^{90} - 64$ is divisible by 91, so the Fermat test fails for $a = 2$, which confirms that 91 is not prime, and $a = 2$ is a witness of this fact.

Unfortunately, the Fermat test has the property that it may succeed for all candidate counterexamples, even though n is composite. The number $n = 561 = 3 \cdot 11 \cdot 17$ is such a devious number. It can be shown that for all $a \in \{2, \ldots, 560\}$ such that $\gcd(a, 561) = 1$, we have

$$a^{560} \equiv 1 \pmod{561},$$

so *all* these a are liars.

Such composite numbers for which the Fermat test succeeds for all candidate counterexamples are called *Carmichael numbers*, and unfortunately there are infinitely many of them. Thus the Fermat test is doomed. There are various ways of strengthening the Fermat test, but we will not discuss this here. We refer the interested reader to Crandall and Pomerance [6] and Gallier and Quaintance [16].

The remedy is to make sure that we pick a property A such that if n is composite, then at least some candidate a is not a liar, and to test all potential countexamples a. The difficulty is that trying all candidate countexamples can be too expensive to be practical.

There are two classes of primality testing algorithms:

(1) Algorithms that try all possible countexamples and for which the test does not lie. These algorithms give a definite answer: n is prime or n is composite. Until 2002, no algorithms running in polynomial time were known. The situation changed in 2002 when a paper with the title "PRIMES is in **P**," by Agrawal, Kayal and Saxena, appeared on the website of the Indian Institute of Technology at Kanpur, India. In this paper, it was shown that testing for primality has a deterministic (nonrandomized) algorithm that runs in polynomial time.

    We will not discuss algorithms of this type here, and instead refer the reader to Crandall and Pomerance [6] and Ribenboim [43].

(2) Randomized algorithms. To avoid having problems with infinite events, we assume that we are testing numbers in some large finite interval $\mathcal{I}$. Given any positive integer $m \in \mathcal{I}$, some candidate witness a is chosen at random. We have a test which, given m and a potential witness a, determines whether or not a is indeed a witness to the fact that m is composite. Such an algorithm is a *Monte Carlo* algorithm, which means the following:

(1) *If the test is positive, then $m \in \mathcal{I}$ is composite.* In terms of probabilities, this is expressed by saying that the conditional probability that $m \in \mathcal{I}$ is composite given that the test is positive is equal to 1. If we denote the event that some positive integer $m \in \mathcal{I}$ is composite by C, then we can express the above as

$$\mathsf{Pr}(C \mid \text{test is positive}) = 1.$$

(2) *If $m \in \mathcal{I}$ is composite, then the test is positive for at least* 50% *of the choices for a.* We can express the above as

$$\mathsf{Pr}(\text{test is positive} \mid C) \geq \frac{1}{2}.$$

This gives us a degree of confidence in the test.

The contrapositive of (1) says that if $m \in \mathcal{I}$ is prime, then the test is negative. If we denote by P the event that some positive integer $m \in \mathcal{I}$ is prime, then this is expressed as

$$\mathsf{Pr}(\text{test is negative} \mid P) = 1.$$

If we repeat the test ℓ times by picking independent potential witnesses, then the conditional probability that the test is negative ℓ times given that n is composite, written $\mathsf{Pr}(\text{test is negative } \ell \text{ times} \mid C)$, is given by

$$\begin{aligned}
\mathsf{Pr}(\text{test is negative } \ell \text{ times} \mid C) &= \mathsf{Pr}(\text{test is negative} \mid C)^{\ell} \\
&= (1 - \mathsf{Pr}(\text{test is positive} \mid C))^{\ell} \\
&\leq \left(1 - \frac{1}{2}\right)^{\ell} \\
&= \left(\frac{1}{2}\right)^{\ell},
\end{aligned}$$

where we used Property (2) of a Monte Carlo algorithm that

$$\mathsf{Pr}(\text{test is positive} \mid C) \geq \frac{1}{2}$$

and the independence of the trials. *This confirms that if we run the algorithm ℓ times, then* Pr(*test is negative ℓ times* $\mid C$) *is very small.* In other words, it is very unlikely that the test will lie ℓ times (is negative) given that the number $m \in \mathcal{I}$ is composite.

If the probabilty $\mathsf{Pr}(P)$ of the event P is known, which requires knowledge of the distribution of the primes in the interval $\mathcal{I}$, then the conditional probability

$$\mathsf{Pr}(P \mid \text{test is negative } \ell \text{ times})$$

can be determined using Bayes's rule.

A Monte Carlo algorithm does not give a definite answer. However, if ℓ is large enough (say $\ell = 100$), then the conditional probability that the number n being tested is prime given that the test is negative ℓ times, is very close to 1.

Two of the best known randomized algorithms for primality testing are the *Miller–Rabin test* and the *Solovay–Strassen test*. We will not discuss these methods here, and we refer the reader to Gallier and Quaintance [16].

However, what we will discuss is a nondeterministic algorithm that checks that a number n is prime by guessing a certain kind of tree that we call a Lucas tree (because this algorithm is based on a method due to E. Lucas), and then verifies in polynomial time (in the length $\log_2 n$ of the input given in binary) that this tree constitutes a "proof" that n is indeed prime. This shows that primality testing is in $\mathcal{NP}$, a fact that is not obvious at all. Of course, this is a much weaker result than the AKS algorithm, but the proof that the AKS works in polynomial time (in $\log_2 n$) is much harder.

The Lucas test, and basically all of the primality-testing algorithms, use modular arithmetic and some elementary facts of number theory such as the Euler-Fermat theorem, so we proceed with a review of these concepts.

12.3 Modular Arithmetic, the Groups $\mathbb{Z}/n\mathbb{Z}$, $(\mathbb{Z}/n\mathbb{Z})^*$

Recall the fundamental notion of congruence modulo n and its notation due to Gauss (circa 1802).

Definition 12.2. For any $a, b \in \mathbb{Z}$, we write $a \equiv b \pmod{m}$ iff $a - b = km$, for some $k \in \mathbb{Z}$ (in other words, $a - b$ is divisible by m), and we say that *a and b are congruent modulo m.*

For example, $37 \equiv 1 \pmod 9$, since $37 - 1 = 36 = 4 \cdot 9$. It can also be shown that $200^{250} \equiv 1 \pmod{251}$, but this is impossible to do by brute force, so we will develop some tools to either avoid such computations, or to make them tractable.

It is easy to check that congruence is an equivalence relation but it also satisfies the following properties.

Proposition 12.3. *For any positive integer m, for all $a_1, a_2, b_1, b_2 \in \mathbb{Z}$, the following properties hold. If $a_1 \equiv b_1 \pmod{m}$ and $a_2 \equiv b_2 \pmod{m}$, then*

(1) $a_1 + a_2 \equiv b_1 + b_2 \pmod{m}$.

(2) $a_1 - a_2 \equiv b_1 - b_2 \pmod{m}$.

(3) $a_1a_2 \equiv b_1b_2 \pmod{m}$.

Proof. We only check (3), leaving (1) and (2) as easy exercises. Because $a_1 \equiv b_1 \pmod{m}$ and $a_2 \equiv b_2 \pmod{m}$, we have $a_1 = b_1 + k_1m$ and $a_2 = b_2 + k_2m$, for some $k_1, k_2 \in \mathbb{Z}$, so we obtain

$$\begin{aligned} a_1a_2 - b_1b_2 &= a_1(a_2 - b_2) + (a_1 - b_1)b_2 \\ &= (a_1k_2 + k_1b_2)m. \end{aligned}$$ ☐

Proposition 12.3 allows us to define addition, subtraction, and multiplication on equivalence classes modulo m.

Definition 12.3. Given any positive integer m, we denote by $\mathbb{Z}/m\mathbb{Z}$ the set of equivalence classes modulo m. If we write $\overline{a}$ for the equivalence class of $a \in \mathbb{Z}$, then we define addition, subtraction, and multiplication on residue classes as follows:

$$\begin{aligned} \overline{a} + \overline{b} &= \overline{a + b} \\ \overline{a} - \overline{b} &= \overline{a - b} \\ \overline{a} \cdot \overline{b} &= \overline{ab}. \end{aligned}$$

The above operations make sense because $\overline{a + b}$ does not depend on the representatives chosen in the equivalence classes $\overline{a}$ and $\overline{b}$, and similarly for $\overline{a - b}$ and $\overline{ab}$. Each equivalence class $\overline{a}$ contains a unique representative from the set of remainders $\{0, 1, \ldots, m-1\}$, modulo m, so the above operations are completely determined by $m \times m$ tables. Using the arithmetic operations of $\mathbb{Z}/m\mathbb{Z}$ is called *modular arithmetic*.

The addition tables of $\mathbb{Z}/n\mathbb{Z}$ for $n = 2, 3, 4, 5, 6, 7$ are shown below.

$n = 2$

+	0	1
0	0	1
1	1	0

$n = 3$

+	0	1	2
0	0	1	2
1	1	2	0
2	2	0	1

$n = 4$

+	0	1	2	3
0	0	1	2	3
1	1	2	3	0
2	2	3	0	1
3	3	0	1	2

$n = 5$

+	0	1	2	3	4
0	0	1	2	3	4
1	1	2	3	4	0
2	2	3	4	0	1
3	3	4	0	1	2
4	4	0	1	2	3

$n = 6$

+	0	1	2	3	4	5
0	0	1	2	3	4	5
1	1	2	3	4	5	0
2	2	3	4	5	0	1
3	3	4	5	0	1	2
4	4	5	0	1	2	3
5	5	0	1	2	3	4

$n = 7$

+	0	1	2	3	4	5	6
0	0	1	2	3	4	5	6
1	1	2	3	4	5	6	0
2	2	3	4	5	6	0	1
3	3	4	5	6	0	1	2
4	4	5	6	0	1	2	3
5	5	6	0	1	2	3	4
6	6	0	1	2	3	4	5

It is easy to check that the addition operation $+$ is commutative (abelian), associative, that 0 is an identity element for $+$, and that every element a has $-a$ as additive inverse, which means that

$$a + (-a) = (-a) + a = 0.$$

The set $\mathbb{Z}/n\mathbb{Z}$ of residue classes modulo n is a group under addition, a notion defined formally in Definition 12.4

It is easy to check that the multiplication operation $\cdot$ is commutative (abelian), associative, that 1 is an identity element for $\cdot$, and that $\cdot$ is distributive on the left and on the right with respect to addition. We usually suppress the dot and write $\overline{a}\,\overline{b}$ instead of $\overline{a}\cdot\overline{b}$. The multiplication tables of $\mathbb{Z}/n\mathbb{Z}$ for $n = 2, 3, \ldots, 9$ are shown below. Since $0 \cdot m = m \cdot 0 = 0$ for all m, these tables are only given for nonzero arguments.

$n = 2$

$\cdot$	1
1	1

$n = 3$

$\cdot$	1	2
1	1	2
2	2	1

$n = 4$

$\cdot$	**1**	2	**3**
1	**1**	2	**3**
2	2	0	2
3	**3**	2	**1**

$n = 5$

$\cdot$	1	2	3	4
1	1	2	3	4
2	2	4	1	3
3	3	1	4	2
4	4	3	2	1

$n = 6$

$\cdot$	**1**	2	3	4	**5**
1	**1**	2	3	4	**5**
2	2	4	0	2	4
3	3	0	3	0	3
4	4	2	0	4	2
5	**5**	4	3	2	**1**

$n = 7$

$\cdot$	1	2	3	4	5	6
1	1	2	3	4	5	6
2	2	4	6	1	3	5
3	3	6	2	5	1	4
4	4	1	5	2	6	3
5	5	3	1	6	4	2
6	6	5	4	3	2	1

$n = 8$

$\cdot$	**1**	2	**3**	4	**5**	6	**7**
1	**1**	2	**3**	4	**5**	6	**7**
2	2	4	6	0	2	4	6
3	**3**	6	**1**	4	**7**	2	**5**
4	4	0	4	0	4	0	4
5	**5**	2	**7**	4	**1**	6	**3**
6	6	4	2	0	6	4	2
7	**7**	6	**5**	4	**3**	2	**1**

$n = 9$

$\cdot$	**1**	**2**	3	**4**	**5**	6	**7**	**8**
1	**1**	**2**	3	**4**	**5**	6	**7**	**8**
2	**2**	**4**	6	**8**	**1**	3	**5**	**7**
3	3	6	0	3	6	0	3	6
4	**4**	**8**	3	**7**	**2**	6	**1**	**5**
5	**5**	**1**	6	**2**	**7**	3	**8**	**4**
6	6	3	0	6	3	0	6	3
7	**7**	**5**	3	**1**	**8**	6	**4**	**2**
8	**8**	**7**	6	**5**	**4**	3	**2**	**1**

Examining the above tables, we observe that for $n = 2, 3, 5, 7$, which are primes, every element has an inverse, which means that for every nonzero element a, there is some (actually, unique) element b such that

$$a \cdot b = b \cdot a = 1.$$

For $n = 2, 3, 5, 7$, the set $\mathbb{Z}/n\mathbb{Z} - \{0\}$ is an abelian group under multiplication (see Definition 12.4). When n is composite, there exist nonzero elements whose product is zero. For example, when $n = 6$, we have $3 \cdot 2 = 0$, when $n = 8$, we have $4 \cdot 4 = 0$, when $n = 9$, we have $6 \cdot 6 = 0$.

For $n = 4, 6, 8, 9$, the elements a that have an inverse are precisely those that are relatively prime to the modulus n (that is, $\gcd(a, n) = 1$).

These observations hold in general. Recall the Bezout theorem: two nonzero integers $m, n \in \mathbb{Z}$ are relatively prime ($\gcd(m, n) = 1$) iff there are integers $a, b \in \mathbb{Z}$ such that

$$am + bn = 1.$$

Proposition 12.4. *Given any integer $n \geq 1$, for any $a \in \mathbb{Z}$, the residue class $\overline{a} \in \mathbb{Z}/n\mathbb{Z}$ is invertible with respect to multiplication iff $\gcd(a, n) = 1$.*

Proof. If $\overline{a}$ has inverse $\overline{b}$ in $\mathbb{Z}/n\mathbb{Z}$, then $\overline{a}\,\overline{b} = 1$, which means that

$$ab \equiv 1 \pmod{n},$$

that is $ab = 1 + nk$ for some $k \in \mathbb{Z}$, which is the Bezout identity

$$ab - nk = 1$$

and implies that $\gcd(a, n) = 1$. Conversely, if $\gcd(a, n) = 1$, then by Bezout's identity there exist $u, v \in \mathbb{Z}$ such that

$$au + nv = 1,$$

so $au = 1 - nv$, that is,

$$au \equiv 1 \pmod{n},$$

which means that $\overline{a}\,\overline{u} = 1$, so $\overline{a}$ is invertible in $\mathbb{Z}/n\mathbb{Z}$. ∎

We have alluded to the notion of a group. Here is the formal definition.

Definition 12.4. A *group* is a set G equipped with a binary operation $\cdot \colon G \times G \to G$ that associates an element $a \cdot b \in G$ to every pair of elements $a, b \in G$, and having the following properties: $\cdot$ is associative, has an identity element $e \in G$, and every element in G is invertible (w.r.t. $\cdot$). More explicitly, this means that the following equations hold for all $a, b, c \in G$:

(G1) $a \cdot (b \cdot c) = (a \cdot b) \cdot c.$ (associativity);

(G2) $a \cdot e = e \cdot a = a.$ (identity);

(G3) For every $a \in G$, there is some $a^{-1} \in G$ such that $a \cdot a^{-1} = a^{-1} \cdot a = e$. (inverse).

A group G is *abelian* (or *commutative*) if

$$a \cdot b = b \cdot a \quad \text{for all } a, b \in G.$$

It is easy to show that the element e satisfying property (G2) is unique, and for any $a \in G$, the element $a^{-1} \in G$ satisfying $a \cdot a^{-1} = a^{-1} \cdot a = e$ required to exist by (G3) is actually unique. This element is called *the* inverse of a.

The set of integers $\mathbb{Z}$ with the addition operation is an abelian group with identity element 0. The set $\mathbb{Z}/n\mathbb{Z}$ of residues modulo m is an abelian group under addition with identity element 0. In general, $\mathbb{Z}/n\mathbb{Z} - \{0\}$ is *not* a group under multiplication, because some nonzero elements may not have an inverse. However, by Proposition 12.4, if p is prime, then $\mathbb{Z}/n\mathbb{Z} - \{0\}$ *is* an abelian group under multiplication.

When p is not prime, the subset of elements, shown in boldface in the multiplication tables, forms an abelian group under multiplication.

Definition 12.5. The group (under multiplication) of invertible elements of the ring $\mathbb{Z}/n\mathbb{Z}$ is denoted by $(\mathbb{Z}/n\mathbb{Z})^*$. Note that this group is abelian and only defined if $n \geq 2$.

Definition 12.6. If G is a finite group, the number of elements in G is called the *the order* of G.

Given a group G with identity element e, and any element $g \in G$, we often need to consider the powers of g defined as follows.

Definition 12.7. Given a group G with identity element e, for any nonnegative integer n, it is natural to define the *power* g^n of g as follows:

$$\begin{aligned} g^0 &= e \\ g^{n+1} &= g \cdot g^n. \end{aligned}$$

Using induction, it is easy to show that

$$g^m g^n = g^{n+m}$$

for all $m, n \in \mathbb{N}$.

Since g has an inverse g^{-1}, we can extend the definition of g^n to negative powers. For $n \in \mathbb{Z}$, with $n < 0$, let

$$g^n = (g^{-1})^{-n}.$$

Then it is easy to prove that

$$
\begin{aligned}
g^i \cdot g^j &= g^{i+j} \\
(g^i)^{-1} &= g^{-i} \\
g^i \cdot g^j &= g^j \cdot g^i
\end{aligned}
$$

for all $i, j \in \mathbb{Z}$.

Given a finite group G of order n, for any element $a \in G$, it is natural to consider the set of powers $\{e, a^1, a^2, \ldots, a^k, \ldots\}$. A crucial fact is that there is a smallest positive $s \in \mathbb{N}$ such that $a^s = e$, and that s divides n.

Proposition 12.5. *Let G be a finite group of order n. For every element $a \in G$, the following facts hold:*

(1) There is a smallest positive integer $s \leq n$ such that $a^s = e$.

(2) The set $\{e, a, \ldots, a^{s-1}\}$ is an abelian group denoted $\langle a \rangle$.

(3) We have $a^n = e$, and the positive integer s divides n, More generally, for any positive integer m, if $a^m = e$, then s divides m.

Proof. (1) Consider the sequence of $n+1$ elements

$$(e, a^1, a^2, \ldots, a^n).$$

Since G only has n distinct elements, by the pigeonhole principle, there exist i, j such that $0 \leq i < j \leq n$ such that

$$a^i = a^j.$$

By multiplying both sides by $(a^i)^{-1} = a^{-i}$, we get

$$e = a^i (a^i)^{-1} = a^j (a^i)^{-1} = a^j a^{-i} = a^{j-i}.$$

Since $0 \leq i < j \leq n$, we have $0 \leq j - i \leq n$ with $a^{j-i} = e$. Thus there is some s with $0 < s \leq n$ such that $a^s = e$, and thus a smallest such s.

(2) Since $a^s = e$, for any $i, j \in \{0, \ldots, s-1\}$ if we write $i + j = sq + r$ with $0 \leq r \leq s-1$, we have

$$a^i a^j = a^{i+j} = a^{sq+r} = a^{sq} a^r = (a^s)^q a^r = e^q a^r = a^r,$$

so $\langle a \rangle$ is closed under multiplication. We have $e \in \langle a \rangle$ and the inverse of a^i is a^{s-i}, so $\langle a \rangle$ is a group. This group is obviously abelian.

(3) For any element $g \in G$, let $g\langle a \rangle = \{ga^k \mid 0 \leq k \leq s-1\}$. Observe that for any $i \in \mathbb{N}$, we have

$$a^i \langle a \rangle = \langle a \rangle.$$

We claim that for any two elements $g_1, g_2 \in G$, if $g_1\langle a \rangle \cap g_2\langle a \rangle \neq \emptyset$, then $g_1\langle a \rangle = g_2\langle a \rangle$.

Proof of the claim. If $g \in g_1\langle a\rangle \cap g_2\langle a\rangle$, then there exist $i, j \in \{0, \ldots, s-1\}$ such that

$$g_1 a^i = g_2 a^j.$$

Without loss of generality, we may assume that $i \geq j$. By multipliying both sides by $(a^j)^{-1}$, we get

$$g_2 = g_1 a^{i-j}.$$

Consequently

$$g_2\langle a\rangle = g_1 a^{i-j}\langle a\rangle = g_1\langle a\rangle,$$

as claimed. ∎

It follows that the pairwise disjoint nonempty subsets of the form $g\langle a\rangle$, for $g \in G$, form a partition of G. However, the map φ_g from $\langle a\rangle$ to $g\langle a\rangle$ given by $\varphi_g(a^i) = ga^i$ has for inverse the map $\varphi_{g^{-1}}$, so φ_g is a bijection, and thus the subsets $g\langle a\rangle$ all have the same number of elements s. Since these subsets form a partition of G, we must have $n = sq$ for some $q \in \mathbb{N}$, which implies that $a^n = e$.

If $g^m = 1$, then writing $m = sq + r$, with $0 \leq r < s$, we get

$$1 = g^m = g^{sq+r} = (g^s)^q \cdot g^r = g^r,$$

so $g^r = 1$ with $0 \leq r < s$, contradicting the minimality of s, so $r = 0$ and s divides m. ∎

Definition 12.8. Given a finite group G of order n, for any $a \in G$, the smallest positive integer $s \leq n$ such that $a^s = e$ in (1) of Proposition 12.5 is called the *order* of a.

The *Euler φ-function* plays an important role in the theory of the groups $(\mathbb{Z}/n\mathbb{Z})^*$.

Definition 12.9. Given any positive integer $n \geq 1$, the *Euler φ-function* (or Euler *totient function*) is defined such that $\varphi(n)$ is the number of integers a, with $1 \leq a \leq n$, which are relatively prime to n; that is, with $\gcd(a, n) = 1$.[1]

If p is prime, then by definition

$$\varphi(p) = p - 1.$$

We leave it as an exercise to show that if p is prime and if $k \geq 1$, then

$$\varphi(p^k) = p^{k-1}(p-1).$$

It can also be shown that if $\gcd(m, n) = 1$, then

$$\varphi(mn) = \varphi(m)\varphi(n).$$

[1] We allow $a = n$ to accomodate the special case $n = 1$.

The above properties yield a method for computing $\varphi(n)$, based on its prime factorization. If $n = p_1^{i_1} \cdots p_k^{i_k}$, then

$$\varphi(n) = p_1^{i_1 - 1} \cdots p_k^{i_k - 1}(p_1 - 1) \cdots (p_k - 1).$$

For example, $\varphi(17) = 16$, $\varphi(49) = 7 \cdot 6 = 42$,

$$\varphi(900) = \varphi(2^2 \cdot 3^2 \cdot 5^2) = 2 \cdot 3 \cdot 5 \cdot 1 \cdot 2 \cdot 4 = 240.$$

Proposition 12.4 shows that $(\mathbb{Z}/n\mathbb{Z})^*$ has $\varphi(n)$ elements. It also shows that $\mathbb{Z}/n\mathbb{Z} - \{0\}$ is a group (under multiplication) iff n is prime.

For any integer $n \geq 2$, let $(\mathbb{Z}/n\mathbb{Z})^*$ be the group of invertible elements of the ring $\mathbb{Z}/n\mathbb{Z}$. This is a group of order $\varphi(n)$. Then Proposition 12.5 yields the following result.

Theorem 12.6. *(Euler) For any integer $n \geq 2$ and any $a \in \{1, \ldots, n-1\}$ such that $\gcd(a, n) = 1$, we have*

$$a^{\varphi(n)} \equiv 1 \pmod{n}.$$

In particular, if n is a prime, then $\varphi(n) = n - 1$, and we get Fermat's little theorem.

Theorem 12.7. *(Fermat's little theorem) For any prime p and any $a \in \{1, \ldots, p-1\}$, we have*

$$a^{p-1} \equiv 1 \pmod{p}.$$

Since 251 is prime, and since $\gcd(200, 252) = 1$, Fermat's little theorem implies our earlier claim that $200^{250} \equiv 1 \pmod{251}$, without making any computations.

Proposition 12.5 suggests considering groups of the form $\langle g \rangle$.

Definition 12.10. A finite group G is *cyclic* iff there is some element $g \in G$ such that $G = \langle g \rangle$. An element $g \in G$ with this property is called a *generator* of G.

Even though, in principle, a finite cyclic group has a very simple structure, finding a generator for a finite cyclic group is generally hard. For example, it turns out that the multiplicative group $(\mathbb{Z}/p\mathbb{Z})^*$ is a cyclic group when p is prime, but no efficient method for finding a generator for $(\mathbb{Z}/p\mathbb{Z})^*$ is known (besides a brute-force search).

Examining the multiplication tables for $(\mathbb{Z}/n\mathbb{Z})^*$ for $n = 3, 4, \ldots, 9$, we can check the following facts:

1. 2 is a generator for $(\mathbb{Z}/3\mathbb{Z})^*$.
2. 3 is a generator for $(\mathbb{Z}/4\mathbb{Z})^*$.
3. 2 is a generator for $(\mathbb{Z}/5\mathbb{Z})^*$.

4. 5 is a generator for $(\mathbb{Z}/6\mathbb{Z})^*$.

5. 3 is a generator for $(\mathbb{Z}/7\mathbb{Z})^*$.

6. Every element of $(\mathbb{Z}/8\mathbb{Z})^*$ satisfies the equation $a^2 = 1 \pmod{8}$, thus $(\mathbb{Z}/8\mathbb{Z})^*$ has no generators.

7. 2 is a generator for $(\mathbb{Z}/9\mathbb{Z})^*$.

More generally, it can be shown that the multiplicative groups $(\mathbb{Z}/p^k\mathbb{Z})^*$ and $(\mathbb{Z}/2p^k\mathbb{Z})^*$ are cyclic groups when p is an odd prime and $k \geq 1$.

Definition 12.11. A generator of the group $(\mathbb{Z}/n\mathbb{Z})^*$ (when there is one), is called a *primitive root modulo n*.

As an exercise, the reader should check that the next value of n for which $(\mathbb{Z}/n\mathbb{Z})^*$ has no generator is $n = 12$.

The following theorem due to Gauss can be shown. For a proof, see Apostol [2] or Gallier and Quaintance [16].

Theorem 12.8. *(Gauss) For every odd prime p, the group $(\mathbb{Z}/p\mathbb{Z})^*$ is cyclic of order $p-1$. It has $\varphi(p-1)$ generators.*

According to Definition 12.11, the generators of $(\mathbb{Z}/p\mathbb{Z})^*$ are the *primitive roots modulo p*.

12.4 The Lucas Theorem

In this section we discuss an application of the existence of primitive roots in $(\mathbb{Z}/p\mathbb{Z})^*$ where p is an odd prime, known an the $n-1$ *test*. This test due to E. Lucas determines whether a positive odd integer n is prime or not by examining the prime factors of $n-1$ and checking some congruences.

The $n-1$ test can be described as the construction of a certain kind of tree rooted with n, and it turns out that the number of nodes in this tree is bounded by $2\log_2 n$, and that the number of modular multiplications involved in checking the congruences is bounded by $2\log_2^2 n$.

When we talk about the complexity of algorithms dealing with numbers, we assume that all inputs (to a Turing machine) are strings representing these numbers, typically in base 2. Since the length of the binary representation of a natural number $n \geq 1$ is $\lfloor \log_2 n \rfloor + 1$ (or $\lceil \log_2(n+1) \rceil$, which allows $n = 0$), the complexity of algorithms dealing with (nonzero) numbers $m, n,$ *etc.* is expressed in terms of $\log_2 m, \log_2 n,$ *etc.* Recall that for any real number $x \in \mathbb{R}$, the *floor of* x is the greatest integer $\lfloor x \rfloor$ that is less that or equal to x, and the *ceiling of* x is the least integer $\lceil x \rceil$ that is greater that or equal to x.

If we choose to represent numbers in base 10, since for any base b we have $\log_b x = \ln x / \ln b$, we have

$$\log_2 x = \frac{\ln 10}{\ln 2} \log_{10} x.$$

Since $(\ln 10)/(\ln 2) \approx 3.322 \approx 10/3$, we see that the number of decimal digits needed to represent the integer n in base 10 is approximately 30% of the number of bits needed to represent n in base 2.

Since the Lucas test yields a tree such that the number of modular multiplications involved in checking the congruences is bounded by $2 \log_2^2 n$, it is not hard to show that testing whether or not a positive integer n is prime, a problem denoted PRIMES, belongs to the complexity class $\mathcal{NP}$. This result was shown by V. Pratt [40] (1975), but Peter Freyd told me that it was "folklore." Since 2002, thanks to the AKS algorithm, we know that PRIMES actually belongs to the class $\mathcal{P}$, but this is a much harder result.

Here is Lehmer's version of the Lucas result, from 1876.

Theorem 12.9. *(Lucas theorem) Let n be a positive integer with $n \geq 2$. Then n is prime iff there is some integer $a \in \{1, 2, \ldots, n-1\}$ such that the following two conditions hold:*

(1) $a^{n-1} \equiv 1 \pmod{n}$.

(2) If $n > 2$, then $a^{(n-1)/q} \not\equiv 1 \pmod{n}$ for all prime divisors q of $n - 1$.

Proof. First assume that Conditions (1) and (2) hold. If $n = 2$, since 2 is prime, we are done. Thus assume that $n \geq 3$, and let r be the order of a (we are working in the abelian group $(\mathbb{Z}/n\mathbb{Z})^*$). We claim that $r = n - 1$. The condition $a^{n-1} \equiv 1 \pmod{n}$ implies that r divides $n-1$. Suppose that $r < n-1$, and let q be a prime divisor of $(n-1)/r$ (so q divides $n - 1$). Since r is the order of a we have $a^r \equiv 1 \pmod{n}$, so we get

$$a^{(n-1)/q} \equiv a^{r(n-1)/(rq)} \equiv (a^r)^{(n-1)/(rq)} \equiv 1^{(n-1)/(rq)} \equiv 1 \pmod{n},$$

contradicting Condition (2). Therefore, $r = n - 1$, as claimed.

We now show that n must be prime. Now $a^{n-1} \equiv 1 \pmod{n}$ implies that a and n are relatively prime so by Euler's theorem (Theorem 12.6),

$$a^{\varphi(n)} \equiv 1 \pmod{n}.$$

Since the order of a is $n - 1$, we have $n - 1 \leq \varphi(n)$. If $n \geq 3$ is not prime, then n has some prime divisor p, but n and p are integers in $\{1, 2, \ldots, n\}$ that are not relatively prime to n, so by definition of $\varphi(n)$, we have $\varphi(n) \leq n - 2$, contradicting the fact that $n - 1 \leq \varphi(n)$. Therefore, n must be prime.

Conversely, assume that n is prime. If $n = 2$, then we set $a = 1$. Otherwise, pick a to be any primitive root modulo p. $\square$

Clearly, if $n > 2$ then we may assume that $a \geq 2$. The main difficulty with the $n-1$ test is not so much guessing the primitive root a, but finding a *complete prime factorization* of $n-1$. However, as a nondeterministic algorithm, the $n-1$ test yields a "proof" that a number n is indeed prime which can be represented as a tree, and the number of operations needed to check the required conditions (the congruences) is bounded by $c \log_2^2 n$ for some positive constant c, and this implies that testing primality is in $\mathcal{NP}$.

Before explaining the details of this method, we sharpen slightly Lucas theorem to deal only with odd prime divisors.

Theorem 12.10. *Let n be a positive odd integer with $n \geq 3$. Then n is prime iff there is some integer $a \in \{2, \ldots, n-1\}$ (a guess for a primitive root modulo n) such that the following two conditions hold:*

(1b) $a^{(n-1)/2} \equiv -1 \pmod{n}$.

(2b) If $n-1$ is not a power of 2, then $a^{(n-1)/2q} \not\equiv -1 \pmod{n}$ for all odd prime divisors q of $n-1$.

Proof. Assume that Conditions (1b) and (2b) of Theorem 12.10 hold. Then we claim that Conditions (1) and (2) of Theorem 12.9 hold. By squaring the congruence $a^{(n-1)/2} \equiv -1 \pmod{n}$, we get $a^{n-1} \equiv 1 \pmod{n}$, which is Condition (1) of Theorem 12.9. Since $a^{(n-1)/2} \equiv -1 \pmod{n}$, Condition (2) of Theorem 12.9 holds for $q = 2$. Next, if q is an odd prime divisor of $n-1$, let $m = a^{(n-1)/2q}$. Condition (1b) means that

$$m^q \equiv a^{(n-1)/2} \equiv -1 \pmod{n}.$$

Now if $m^2 \equiv a^{(n-1)/q} \equiv 1 \pmod{n}$, since q is an odd prime, we can write $q = 2k+1$ for some $k \geq 1$, and then

$$m^q \equiv m^{2k+1} \equiv (m^2)^k m \equiv 1^k m \equiv m \pmod{n},$$

and since $m^q \equiv -1 \pmod{n}$, we get

$$m \equiv -1 \pmod{n}$$

(regardless of whether n is prime or not). Thus we proved that if $m^q \equiv -1 \pmod{n}$ and $m^2 \equiv 1 \pmod{n}$, then $m \equiv -1 \pmod{n}$. By contrapositive, we see that if $m \not\equiv -1 \pmod{n}$, then $m^2 \not\equiv 1 \pmod{n}$ or $m^q \not\equiv -1 \pmod{n}$, but since $m^q \equiv a^{(n-1)/2} \equiv -1 \pmod{n}$ by Condition (1a), we conclude that $m^2 \equiv a^{(n-1)/q} \not\equiv 1 \pmod{n}$, which is Condition (2) of Theorem 12.9. But then Theorem 12.9 implies that n is prime.

Conversely, assume that n is an odd prime, and let a be any primitive root modulo n. Then by little Fermat we know that

$$a^{n-1} \equiv 1 \pmod{n},$$

so

$$(a^{(n-1)/2} - 1)(a^{(n-1)/2} + 1) \equiv 0 \pmod{n}.$$

Since n is prime, either $a^{(n-1)/2} \equiv 1 \pmod{n}$ or $a^{(n-1)/2} \equiv -1 \pmod{n}$, but since a generates $(\mathbb{Z}/n\mathbb{Z})^*$, it has order $n-1$, so the congruence $a^{(n-1)/2} \equiv 1 \pmod{n}$ is impossible, and Condition (1b) must hold. Similarly, if we had $a^{(n-1)/2q} \equiv -1 \pmod{n}$ for some odd prime divisor q of $n-1$, then by squaring we would have

$$a^{(n-1)/q} \equiv 1 \pmod{n},$$

and a would have order at most $(n-1)/q < n-1$, which is absurd. $\square$

12.5 Lucas Trees

If n is an odd prime, we can use Theorem 12.10 to build recursively a tree which is a proof, or certificate, of the fact that n is indeed prime. We first illustrate this process with the prime $n = 1279$.

Example 12.1. If $n = 1279$, then we easily check that $n - 1 = 1278 = 2 \cdot 3^2 \cdot 71$. We build a tree whose root node contains the triple $(1279, ((2,1), (3,2), (71,1)), 3)$, where $a = 3$ is the guess for a primitive root modulo 1279. In this simple example, it is clear that 3 and 71 are prime, but we must supply proofs that these number are prime, so we recursively apply the process to the odd divisors 3 and 71.

Since $3 - 1 = 2^1$ is a power of 2, we create a one-node tree $(3, ((2,1)), 2)$, where $a = 2$ is a guess for a primitive root modulo 3. This is a leaf node.

Since $71-1 = 70 = 2\cdot 5\cdot 7$, we create a tree whose root node is $(71, ((2,1), (5,1), (7,1)), 7)$, where $a = 7$ is the guess for a primitive root modulo 71. Since $5 - 1 = 4 = 2^2$, and $7 - 1 = 6 = 2 \cdot 3$, this node has two successors $(5, ((2,2)), 2)$ and $(7, ((2,1), (3,1)), 3)$, where 2 is the guess for a primitive root modulo 5, and 3 is the guess for a primitive root modulo 7.

Since $4 = 2^2$ is a power of 2, the node $(5, ((2,2)), 2)$ is a leaf node.

Since $3 - 1 = 2^1$, the node $(7, ((2,1), (3,1)), 3)$ has a single successor, $(3, ((2,1)), 2)$, where $a = 2$ is a guess for a primitive root modulo 3. Since $2 = 2^1$ is a power of 2, the node $(3, ((2,1)), 2)$ is a leaf node.

To recap, we obtain the following tree:

$$
\begin{tikzcd}
 & \begin{array}{c}(1279,\\ ((2,1),(3,2),(71,1)),3)\end{array} \arrow[dl] \arrow[dr] & & \\
\begin{array}{c}(3,\\ ((2,1)),2)\end{array} & & \begin{array}{c}(71,\\ ((2,1),(5,1),(7,1)),7)\end{array} \arrow[dl] \arrow[dr] & \\
 & \begin{array}{c}(5,\\ ((2,2)),2)\end{array} & & \begin{array}{c}(7,\\ ((2,1),(3,1)),3)\end{array} \arrow[d] \\
 & & & \begin{array}{c}(3,\\ ((2,1)),2)\end{array}
\end{tikzcd}
$$

We still have to check that the relevant congruences hold at every node. For the root node $(1279, ((2,1),(3,2),(71,1)),3)$, we check that

$$3^{1278/2} \equiv 3^{864} \equiv -1 \pmod{1279} \tag{1b}$$
$$3^{1278/(2\cdot 3)} \equiv 3^{213} \equiv 775 \pmod{1279} \tag{2b}$$
$$3^{1278/(2\cdot 71)} \equiv 3^{9} \equiv 498 \pmod{1279}. \tag{2b}$$

Assuming that 3 and 71 are prime, the above congruences check that Conditions (1a) and (2b) are satisfied, and by Theorem 12.10 this proves that 1279 is prime. We still have to certify that 3 and 71 are prime, and we do this recursively.

For the leaf node $(3, ((2,1)),2)$, we check that

$$2^{2/2} \equiv -1 \pmod{3}. \tag{1b}$$

For the node $(71, ((2,1),(5,1),(7,1)),7)$, we check that

$$7^{70/2} \equiv 7^{35} \equiv -1 \pmod{71} \tag{1b}$$
$$7^{70/(2\cdot 5)} \equiv 7^{7} \equiv 14 \pmod{71} \tag{2b}$$
$$7^{70/(2\cdot 7)} \equiv 7^{5} \equiv 51 \pmod{71}. \tag{2b}$$

Now we certified that 3 and 71 are prime, assuming that 5 and 7 are prime, which we now establish.

For the leaf node $(5, ((2,2)),2)$, we check that

$$2^{4/2} \equiv 2^{2} \equiv -1 \pmod{5}. \tag{1b}$$

For the node $(7,((2,1),(3,1)),3)$, we check that

$$3^{6/2} \equiv 3^3 \equiv -1 \pmod 7 \tag{1b}$$

$$3^{6/(2\cdot 3)} \equiv 3^1 \equiv 3 \pmod 7. \tag{2b}$$

We have certified that 5 and 7 are prime, given that 3 is prime, which we finally verify.

At last, for the leaf node $(3,((2,1)),2)$, we check that

$$2^{2/2} \equiv -1 \pmod 3. \tag{1b}$$

The above example suggests the following definition.

Definition 12.12. Given any odd integer $n \geq 3$, a *pre-Lucas tree for* n is defined inductively as follows:

(1) It is a one-node tree labeled with $(n,((2,i_0)),a)$, such that $n-1 = 2^{i_0}$, for some $i_0 \geq 1$ and some $a \in \{2,\ldots,n-1\}$.

(2) If $L_1,\ldots,L_k$ are k pre-Lucas (with $k \geq 1$), where the tree L_j is a pre-Lucas tree for some odd integer $q_j \geq 3$, then the tree L whose root is labeled with $(n,((2,i_0),((q_1,i_1),\ldots,$ $(q_k,i_k)),a)$ and whose jth subtree is L_j is a *pre-Lucas tree* for n if

$$n-1 = 2^{i_0} q_1^{i_1} \cdots q_k^{i_k},$$

for some $i_0,i_1,\ldots,i_k \geq 1$, and some $a \in \{2,\ldots,n-1\}$.

Both in (1) and (2), the number a is a guess for a primitive root modulo n.

A pre-Lucas tree for n is a *Lucas tree for* n if the following conditions are satisfied:

(3) If L is a one-node tree labeled with $(n,((2,i_0)),a)$, then

$$a^{(n-1)/2} \equiv -1 \pmod n.$$

(4) If L is a pre-Lucas tree whose root is labeled with $(n,((2,i_0),((q_1,i_1),\ldots,(q_k,i_k)),a)$, and whose jth subtree L_j is a pre-Lucas tree for q_j, then L_j is a Lucas tree for q_j for $j=1,\ldots,k$, and

(a) $a^{(n-1)/2} \equiv -1 \pmod n$.

(b) $a^{(n-1)/2q_j} \not\equiv -1 \pmod n$ for $j=1,\ldots,k$.

Since Conditions (3) and (4) of Definition 12.12 are Conditions (1b) and (2b) of Theorem, 12.10, we see that Definition 12.12 has been designed in such a way that Theorem 12.10 yields the following result.

Theorem 12.11. *An odd integer $n \geq 3$ is prime iff it has some Lucas tree.*

The issue is now to see how long it takes to check that a pre-Lucas tree is a Lucas tree. For this, we need a method for computing $x^n \bmod n$ in polynomial time in $\log_2 n$. This is the object of the next section.

12.6 Algorithms for Computing Powers Modulo m

Let us first consider computing the nth power x^n of some positive integer. The idea is to look at the parity of n and to proceed recursively. If n is even, say $n = 2k$, then

$$x^n = x^{2k} = (x^k)^2,$$

so, compute x^k recursively and then square the result. If n is odd, say $n = 2k+1$, then

$$x^n = x^{2k+1} = (x^k)^2 \cdot x,$$

so, compute x^k recursively, square it, and multiply the result by x.

What this suggests is to write $n \geq 1$ in binary, say

$$n = b_\ell \cdot 2^\ell + b_{\ell-1} \cdot 2^{\ell-1} + \cdots + b_1 \cdot 2^1 + b_0,$$

where $b_i \in \{0, 1\}$ with $b_\ell = 1$ or, if we let $J = \{j \mid b_j = 1\}$, as

$$n = \sum_{j \in J} 2^j.$$

Then we have

$$x^n \equiv x^{\sum_{j \in J} 2^j} = \prod_{j \in J} x^{2^j} \bmod m.$$

This suggests computing the residues r_j such that

$$x^{2^j} \equiv r_j \pmod{m},$$

because then,

$$x^n \equiv \prod_{j \in J} r_j \pmod{m},$$

where we can compute this latter product modulo m two terms at a time.

For example, say we want to compute $999^{179} \bmod 1763$. First, we observe that

$$179 = 2^7 + 2^5 + 2^4 + 2^1 + 1,$$

and we compute the powers modulo 1763:

$$\begin{aligned}
999^{2^1} &\equiv 143 \pmod{1763} \\
999^{2^2} &\equiv 143^2 \equiv 1056 \pmod{1763} \\
999^{2^3} &\equiv 1056^2 \equiv 920 \pmod{1763} \\
999^{2^4} &\equiv 920^2 \equiv 160 \pmod{1763} \\
999^{2^5} &\equiv 160^2 \equiv 918 \pmod{1763} \\
999^{2^6} &\equiv 918^2 \equiv 10 \pmod{1763} \\
999^{2^7} &\equiv 10^2 \equiv 100 \pmod{1763}.
\end{aligned}$$

Consequently,

$$
\begin{aligned}
999^{179} &\equiv 999 \cdot 143 \cdot 160 \cdot 918 \cdot 100 \pmod{1763} \\
&\equiv 54 \cdot 160 \cdot 918 \cdot 100 \pmod{1763}
\end{aligned}
$$

$$
\begin{aligned}
&\equiv 1588 \cdot 918 \cdot 100 \pmod{1763} \\
&\equiv 1546 \cdot 100 \pmod{1763} \\
&\equiv 1219 \pmod{1763},
\end{aligned}
$$

and we find that

$$999^{179} \equiv 1219 \pmod{1763}.$$

Of course, it would be impossible to exponentiate 999^{179} first and then reduce modulo 1763. As we can see, the number of multiplications needed is bounded by $2\log_2 n$, which is quite good.

The above method can be implemented without actually converting n to base 2. If n is even, say $n = 2k$, then $n/2 = k$, and if n is odd, say $n = 2k + 1$, then $(n-1)/2 = k$, so we have a way of dropping the unit digit in the binary expansion of n and shifting the remaining digits one place to the right without explicitly computing this binary expansion. Here is an algorithm for computing $x^n \bmod m$, with $n \geq 1$, using the *repeated squaring* method.

An Algorithm to Compute $x^n \bmod m$ Using Repeated Squaring

begin
  $u := 1$; $a := x$;
  while $n > 1$ **do**
    if even(n) **then** $e := 0$ **else** $e := 1$;
    if $e = 1$ **then** $u := a \cdot u \bmod m$;
    $a := a^2 \bmod m$; $n := (n - e)/2$
  endwhile;
  $u := a \cdot u \bmod m$
end

The final value of u is the result. The reason why the algorithm is correct is that after j rounds through the while loop, $a = x^{2^j} \bmod m$ and

$$u = \prod_{i \in J \mid i < j} x^{2^i} \bmod m,$$

with this product interpreted as 1 when $j = 0$.

Observe that the while loop is only executed $n-1$ times to avoid squaring once more unnecessarily and the last multiplication $a \cdot u$ is performed outside of the while loop. Also, if we delete the reductions modulo m, the above algorithm is a fast method for computing the nth power of an integer x and the time speed-up of not performing the last squaring step is more significant. We leave the details of the proof that the above algorithm is correct as an exercise.

12.7 PRIMES is in $\mathcal{NP}$

Exponentiation modulo n can performed by repeated squaring, as explained in Section 12.6. In that section, we observed that computing $x^m \bmod n$ requires at most $2\log_2 m$ modular multiplications. Using this fact, we obtain the following result adapted from Crandall and Pomerance [6].

Proposition 12.12. *If p is any odd prime, then any pre-Lucas tree L for p has at most $\log_2 p$ nodes, and the number $M(p)$ of modular multiplications required to check that the pre-Lucas tree L is a Lucas tree is less than $2\log_2^2 p$.*

Proof. Let $N(p)$ be the number of nodes in a pre-Lucas tree for p. We proceed by complete induction. If $p = 3$, then $p - 1 = 2^1$, any pre-Lucas tree has a single node, and $1 < \log_2 3$.

Suppose the results holds for any odd prime less than p. If $p - 1 = 2^{i_0}$, then any Lucas tree has a single node, and $1 < \log_2 3 < \log_2 p$. If $p - 1$ has the prime factorization

$$p - 1 = 2^{i_0} q_1^{i_1} \cdots q_k^{i_k},$$

then by the induction hypothesis, each pre-Lucas tree L_j for q_j has less than $\log_2 q_j$ nodes, so

$$N(p) = 1 + \sum_{j=1}^{k} N(q_j) < 1 + \sum_{j=1}^{k} \log_2 q_j = 1 + \log_2(q_1 \cdots q_k) \leq 1 + \log_2\left(\frac{p-1}{2}\right) < \log_2 p,$$

establishing the induction hypothesis.

If r is one of the odd primes in the pre-Lucas tree for p, and $r < p$, then there is some other odd prime q in this pre-Lucas tree such that r divides $q - 1$ and $q \leq p$. We also have to show that at some point, $a^{(q-1)/2r} \not\equiv -1 \pmod{q}$ for some a, and at another point, that $b^{(r-1)/2} \equiv -1 \pmod{r}$ for some b. Using the fact that the number of modular multiplications required to exponentiate to the power m is at most $2\log_2 m$, we see that the number of multiplications required by the above two exponentiations does not exceed

$$2\log_2\left(\frac{q-1}{2r}\right) + 2\log_2\left(\frac{r-1}{2}\right) = 2\log_2\left(\frac{(q-1)(r-1)}{4r}\right) < 2\log_2 q - 4 < 2\log_2 p.$$

As a consequence, we have

$$M(p) < 2\log_2\left(\frac{p-1}{2}\right) + (N(p)-1)2\log_2 p < 2\log_2 p + (\log_2 p - 1)2\log_2 p = 2\log_2^2 p,$$

as claimed. ☐

The following impressive example is from Pratt [40].

Example 12.2. Let $n = 474\,397\,531$. It is easy to check that $n - 1 = 474\,397\,531 - 1 = 474\,397\,530 = 2 \cdot 3 \cdot 5 \cdot 251^3$. We claim that the following is a Lucas tree for $n = 474\,397\,531$:

$$\begin{tikzcd}
& (474\,397\,531, ((2,1),(3,1),(5,1),(251,3)), 2) \arrow[dl] \arrow[d] \arrow[dr] & \\
(3, ((2,1)), 2) & (5, ((2,2)), 2) & (251, ((2,1),(5,3)), 6) \arrow[d] \\
& & (5, ((2,2)), 2)
\end{tikzcd}$$

To verify that the above pre-Lucas tree is a Lucas tree, we check that 2 is indeed a primitive root modulo 474 397 531 by computing (using `Mathematica`) that

$$\begin{aligned}
2^{474\,397\,530/2} &\equiv 2^{237\,198\,765} \equiv -1 \pmod{474\,397\,531} && (1)\\
2^{474\,397\,530/(2\cdot 3)} &\equiv 2^{79\,066\,255} \equiv 9\,583\,569 \pmod{474\,397\,531} && (2)\\
2^{474\,397\,530/(2\cdot 5)} &\equiv 2^{47\,439\,753} \equiv 91\,151\,207 \pmod{474\,397\,531} && (3)\\
2^{474\,397\,530/(2\cdot 251)} &\equiv 2^{945\,015} \equiv 282\,211\,150 \pmod{474\,397\,531}. && (4)
\end{aligned}$$

The number of modular multiplications is: 27 in (1), 26 in (2), 25 in (3) and 19 in (4).

We have $251 - 1 = 250 = 2 \cdot 5^3$, and we verify that 6 is a primitive root modulo 251 by computing:

$$\begin{aligned}
6^{250/2} &\equiv 6^{125} \equiv -1 \pmod{251} && (5)\\
6^{250/(2\cdot 5)} &\equiv 6^{10} \equiv 175 \pmod{251}. && (6)
\end{aligned}$$

The number of modular multiplications is: 6 in (5), and 3 in (6).

We have $5 - 1 = 4 = 2^2$, and 2 is a primitive root modulo 5, since

$$2^{4/2} \equiv 2^2 \equiv -1 \pmod{5}. \qquad (7)$$

This takes one multiplication.

We have $3 - 1 = 2 = 2^1$, and 2 is a primitive root modulo 3, since

$$2^{2/2} \equiv 2^1 \equiv -1 \pmod{3}. \qquad (8)$$

This takes 0 multiplications.

Therefore, 474 397 531 is prime.

As nice as it is, Proposition 12.12 is deceiving, because *finding* a Lucas tree is hard.

Remark: Pratt [40] presents his method for finding a certificate of primality in terms of a proof system. Although quite elegant, we feel that this method is not as transparent as the method using Lucas trees, which we adapted from Crandall and Pomerance [6]. Pratt's proofs can be represented as trees, as Pratt sketches in Section 3 of his paper. However, Pratt uses the basic version of Lucas' theorem, Theorem 12.9, instead of the improved version, Theorem 12.10, so his proof trees have at least twice as many nodes as ours.

As nice as it is, Proposition 12.12 is deceiving, because *finding* a Lucas tree is hard.

The following nice result was first shown by V. Pratt in 1975 [40].

Theorem 12.13. *The problem* PRIMES *(testing whether an integer is prime) is in $\mathcal{NP}$.*

Proof. Since all even integers besides 2 are composite, we can restrict out attention to odd integers $n \geq 3$. By Theorem 12.11, an odd integer $n \geq 3$ is prime iff it has a Lucas tree. Given any odd integer $n \geq 3$, since all the numbers involved in the definition of a pre-Lucas tree are less than n, there is a finite (very large) number of pre-Lucas trees for n. Given a guess of a Lucas tree for n, checking that this tree is a pre-Lucas tree can be performed in $O(\log_2 n)$, and by Proposition 12.12, checking that it is a Lucas tree can be done in $O(\log_2^2 n)$. Therefore PRIMES is in $\mathcal{NP}$. $\square$

Of course, checking whether a number n is composite is in $\mathcal{NP}$, since it suffices to guess to factors n_1, n_2 and to check that $n = n_1 n_2$, which can be done in polynomial time in $\log_2 n$. Therefore, PRIMES $\in \mathcal{NP} \cap \mathrm{co}\mathcal{NP}$. As we said earlier, this was the situation until the discovery of the AKS algorithm, which places PRIMES in $\mathcal{P}$.

Remark: Altough finding a primitive root modulo p is hard, we know that the number of primitive roots modulo p is $\varphi(\varphi(p))$. If p is large enough, this number is actually quite large. According to Crandal and Pomerance [6] (Chapter 4, Section 4.1.1), if p is a prime and if $p > 200560490131$, then p has more than $p/(2 \ln \ln p)$ primitive roots.

Chapter 13

Polynomial- Space Complexity; $\mathcal{PS}$ and $\mathcal{NPS}$

13.1 The Classes $\mathcal{PS}$ (or PSPACE) and $\mathcal{NPS}$ (NPSPACE)

In this chapter we consider complexity classes based on restricting the amount of *space* used by the Turing machine rather than the amount of time.

Definition 13.1. A deterministic or nondeterminitic Turing machine M is *polynomial-space bounded* if there is a polynomial $p(X)$ such that for every input $x \in \Sigma^*$, no matter how much time it uses, the machine M never visits more than $p(|x|)$ tape cells (symbols). Equivalently, for every ID $upav$ arising during the computation, we have $|uav| \leq p(|x|)$.

The class of languages $L \subseteq \Sigma^*$ accepted by some *deterministic* polynomial-space bounded Turing machine is denoted by $\mathcal{PS}$ or PSPACE. Similarly, the class of languages $L \subseteq \Sigma^*$ accepted by some *nondeterministic* polynomial-space bounded Turing machine is denoted by $\mathcal{NPS}$ or NPSPACE.

Obviously $\mathcal{PS} \subseteq \mathcal{NPS}$. Since a (time) polynomially bounded Turing machine can't visit more tape cells (symbols) than one plus the number of moves it makes, we have

$$\mathcal{P} \subseteq \mathcal{PS} \quad \text{and} \quad \mathcal{NP} \subseteq \mathcal{NPS}.$$

Nobody knows whether these inclusions are strict, but this is the most likely assumption. Unlike the situation for time-bounded Turing machines where the big open problem is whether $\mathcal{P} \neq \mathcal{NP}$, for *time-bounded* Turing machines, we have

$$\mathcal{PS} = \mathcal{NPS}.$$

Walter Savitch proved this result iin 1970 (and it is known as *Savitch's theorem*).

Now Definition 13.1 does not say anything about the time-complexity of the Turing machine, so such a machine could even run forever. However, the number of ID's that a polynomial-space bounded Turing machine can visit started on input x is a function of $|x|$ of the form $sp(|x|)t^{p(|x|)}$ for some constants $s > 0$ and $t > 0$, so by the pigeonhole principle, it the number of moves is larger than a certain constant ($c^{1+p(|x|)}$ with $c = s + t$), then some ID must repeat. This fact can be used to show that there is a shorter computation accepting x of length at most $c^{1+p(|x|)}$.

Proposition 13.1. *For any deterministic or nondeterministic polynomial-space bouned Turing machine M with polynomal space bound $p(X)$, there is a constant $c > 1$ such that for every input $x \in \Sigma^*$, if M accepts x, then M accepts x in at most $c^{1+p(|x|)}$ steps.*

Proof. Suppose there are t symbols in the tape alphabet and s states. Then the number of distinct ID's when only $p(|x|)$ tape cells are used is at most $sp(|x|)t^{p(|x|)}$, because we can choose one of s states, place the reading head in any of $p(|x|)$ distinct positions, and there are $t^{p(|x|)}$ strings of tape symbols of length $p(|x|)$. If we let $c = s + t$, by the binomial formula we have

$$\begin{aligned} c^{1+p(|x|)} = (s+t)^{1+p(|x|)} &= \sum_{k=0}^{1+p(|x|)} \binom{1+p(|x|)}{k} s^k t^{1+p(|x|)-k)} \\ &= t^{1+p(|x|)} + (1+p(|x|))st^{p(|x|)} + \cdots \end{aligned}$$

Obviously $(1+p(|x|))st^{p(|x|)} > sp(|x|)t^{p(|x|)}$, so if the number of ID's in the computation is greater than $c^{1+p(|x|)}$, by the pigeonhole principle, two ID's must be identical. By considering a shortest accepting sequence of ID's with n steps, we deduce that $n \leq c^{1+p(|x|)}$, since otherwise the preceding argument shows that the computation would be of the form

$$ID_0 \vdash^* \cdots \vdash^* ID_h \vdash^+ ID_k \vdash^* ID_n$$

with $ID_h = ID_k$, so we would have an even shorter computation

$$ID_0 \vdash^* \cdots \vdash^* ID_h \vdash^* ID_n,$$

contradicting the minimality of the original computation. $\square$

Proposition 13.1 implies that languages in $\mathcal{NPS}$ are computable (in fact, primitive recursive, and even in $\mathcal{EXP}$). This still does not show that languages in $\mathcal{NPS}$ are accepted by polynomial-space Turing machines that *always halt* within some time $c^{q(|x|)}$ for some polynomial $q(X)$. Such a result can be shown using a simulation involving a Turing machine with two tapes.

Proposition 13.2. *For any language $L \in \mathcal{PS}$ (resp. $L \in \mathcal{NPS}$), there is deterministic (resp. nondeterministic) polynomial-space bounded Turing machine M, a polynomal $q(X)$ and a constant $c > 1$, such that for every input $x \in \Sigma^*$, M accepts x in at most $c^{q(|x|)}$ steps.*

A proof of Proposition 13.2 can be found in Hopcroft, Motwani and Ullman [29] (Section 11.2.2, Theorem 11.4).

We now turn to Savitch's theorem.

13.2 Savitch's Theorem: $\mathcal{PS} = \mathcal{NPS}$

The key to the fact that $\mathcal{PS} = \mathcal{NPS}$ is that given a polynomial-space bounded nondeterministic Turing machine M, there is a recursive method to check whether $I \vdash^k J$ with $0 \leq k \leq m$ using at *most* $\log_2 m$ recursive calls, for any two ID's I and J and any natural number $m \geq 1$, (that is, whether there is some computation of $k \leq m$ steps from I to J).

The idea is reminiscent of binary search, namely, to recursively find some intermediate ID K such that $I \vdash^{m_1} K$ and $K \vdash^{m_2} J$ with $m_1 \leq m/2$ and $m_2 \leq m/2$ (here $m/2$ is the integer quotient obtained by dividing m by 2). Because the Turing machine M is polynomial-space bounded, for a given input x, we know from Proposition 13.1 that there are at most $c^{1+p(|x|)}$ distinct ID's, so the search is finite. We will intitially set $m = c^{1+p(|x|)}$, so at most $\log_2 c^{1+p(|x|)} = O(p(|x|)$ recursive calls will be made. We will show that each stack frame takes $O(p(|x|)$ space, so altogether the search uses $O(p(|x|)^2)$ amount of space. This is the crux of Savitch's argument.

The recursive procedure that deals with stack frames of the form $[I, J, m]$ is shown below.

function $reach(I, J, m)$: **boolean**
begin
  if $m = 1$ **then**
    if $I = J = K$ **or** $I \vdash^1 J$ **then**
        $reach =$ **true**
    else
        $reach =$ **false**
    endif
  else
    for each possible ID K **do**
      if $reach(I, K, m/2)$ **and** $reach(K, J, m/2)$ **then**
        $reach =$ **true**
      else
        $reach =$ **false**
      endif
    endfor
  endif
end

Even though the above procedure makes two recursive calls, they are performed sequentially, so the maximum number of stack frames that may arise corresponds to the sequence

$$[I_1, J_1, m],\ [I_2, J_2, m/2],\ [I_3, J_3, m/4],\ [I_4, J_4, m/8],\ \cdots, [I_k, J_k, m/2^{k-1}], \cdots$$

which has length at most $\log_2 m$. Using the procedure *search*, we obtain Savitch's theorem.

Theorem 13.3. *(Savitch, 1970) The complexity classes $\mathcal{PS}$ and $\mathcal{NPS}$ are identical. In fact, if L is accepted by the polynomial-space bounded nondeterministic Turing machine M with space bound $p(X)$, then there is a polynomial-space bounded deterministic Turing machine D accepting L with space bound $O(p(X)^2)$.*

Sketch of proof. Assume that L is accepted by the polynomial-space bounded nondeterministic Turing machine M with space bound $p(X)$. By Proposition 13.1 we may assume that M accepts any input $x \in L$ in at most $c^{1+p(|x|)}$ steps (for some $c > 1$). Set $m = c^{1+p(|x|)}$.

We can design a deterministic Turing machine D which determines (using the function *search*) whether $I_0 \vdash^k J$ with $k \leq m$ where $I_0 = q_0x$ is the starting ID, for all accepting ID's J, by enumerate all accepting ID's J using at most $p(|x|)$ tape cells, using a scratch tape.

As we explained above, the function *search* makes no more than $\log_2 c^{1+p(|x|)} = O(p(|x|)$ recursive calls, Each stack frame takes $O(p(|x|)$ space. The reason is that every ID has at most $1 + p(|x|)$ tape cells and that if we write $m = c^{1+p(|x|)}$ in binary, this takes $\log_2 m = O(p(|x|)$ tape cells. Since at most $O(p(|x|)$ stack frames may arise and since each stack frame has size at most $O(p(|x|)$, the deterministic TM D uses at most $O(p(|x|)^2$ space. For more details, see Hopcroft, Motwani and Ullman [29] (Section 11.2.3, Theorem 11.5). $\square$

Savitch's theorem and Proposition 13.1 show that $\mathcal{PS} = \mathcal{NPS} \subseteq \mathcal{EXP}$. Whether this inclusion is strict is an open problem. The present status of the relative containments of the complexity classes that we have discussed so far is illustrated in Figure 13.1

Savitch's theorem shows that nondeterminism does not help as far as polynomial *space* is concerned, but we still don't have a good example of a language in $\mathcal{PS} = \mathcal{NPS}$ which is not known to be in $\mathcal{NP}$. The next section is devoted to such a problem. This problem also turns out to be $\mathcal{PS}$-complete, so we discuss this notion as well.

13.3 A Complete Problem for $\mathcal{PS}$: QBF

Logic is a natural source of problems complete with respect to a number of complexity classes: SAT is $\mathcal{NP}$-complete (see Theorem 10.8), TAUT is co$\mathcal{NP}$-complete (see Proposition 11.3). It turns out that the validity problem for quantified boolean formulae is $\mathcal{PS}$-complete. We will describe this problem shortly, but first we define $\mathcal{PS}$-completeness.

Definition 13.2. A language $L \subseteq \Sigma^*$ is $\mathcal{PS}$-complete if:

(1) $L \in \mathcal{PS}$.

(2) For every language $L_2 \in \mathcal{PS}$, there is a *polynomial-time computable* function $f\colon \Sigma^* \to \Sigma^*$ such that $x \in L_2$ iff $f(x) \in L$, for all $x \in \Sigma^*$.

Observe that we require the reduction function f to be *polynomial-time computable* rather than *polynomial-space computable*. The reason for this is that with this stronger form of reduction we can prove the following proposition whose simple proof is left as an exercise.

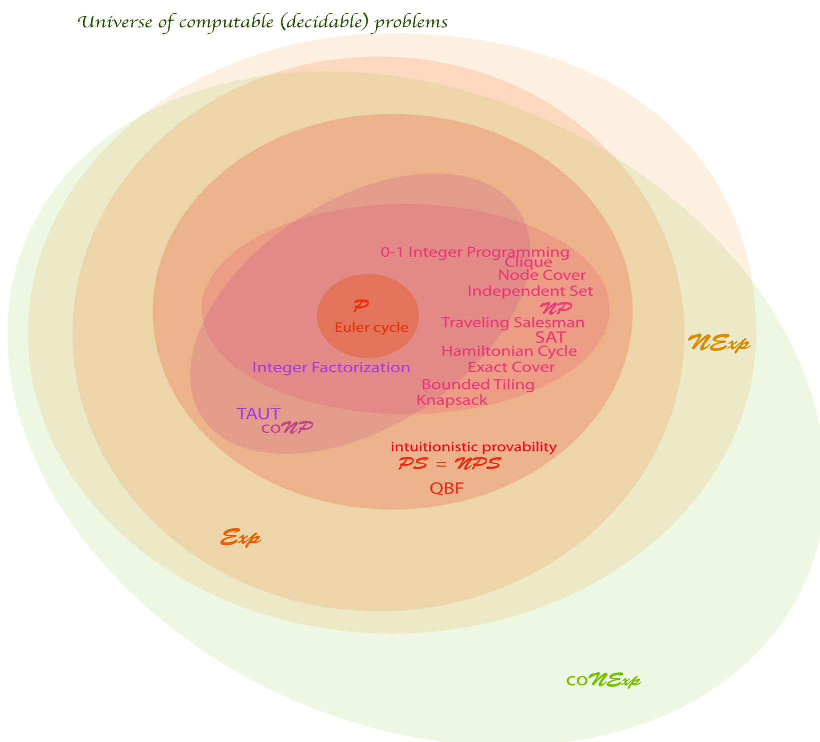


Figure 13.1: Relative containments of the complexity classes.

Proposition 13.4. *Suppose L is a $\mathcal{PS}$-complete language. Then the following facts hold:*

(1) If $L \in \mathcal{P}$, then $\mathcal{P} = \mathcal{PS}$.

(2) If $L \in \mathcal{NP}$, then $\mathcal{NP} = \mathcal{PS}$.

The premises in Proposition 13.4 are very unlikely, but we never know!

We now define the class of quantified boolean formulae. These are actually second-order formulae because we are allowed to quantify over propositional variables, which are 0-ary (constant) predicate symbols. As we will see, validity is still decidable, but the fact that we allow alternation of the quantifiers $\forall$ and $\exists$ makes the problem harder, in the sense that testing validity or nonvalidity no longer appears to be doable in $\mathcal{NP}$ (so far, nobody knows how to do this!).

Recall from Section 10.5 that we have a countable set **PV** of *propositional (or boolean)* *variables*,

$$\mathbf{PV} = \{x_1, x_2, \ldots, \}.$$

Definition 13.3. A *quantified boolean formula* (for short *QBF*) is an expression A defined inductively as follows:

(1) The constants $\top$ and $\perp$ and every propositional variable x_i are QBF's called *atomic* QBF's.

(2) If B is a QBF, then $\neg B$ is a QBF.

(3) If B and C are QBF's, then $(B \vee C)$ is a QBF.

(4) If B and C are QBF's, then $(B \wedge C)$ is a QBF.

(5) If B is a QBF and if x is a propositional variable, then $\forall x B$ is a QBF. The variable x is said to be *universally bound by* $\forall$.

(6) If B is a QBF and if x is a propositional variable, then $\exists x B$ is a QBF. The variable x is said to be *existentially bound by* $\exists$.

(7) If allow the connective $\Rightarrow$, and if B and C are QBF's, then $(B \Rightarrow C)$ is a QBF.

Example 13.1. The following formula is a QBF:

$$A = \forall x\big(\exists y(x \wedge y) \vee \forall z(\neg x \vee z)\big).$$

As usual, we can define inductively the notion of free and bound variable as follows.

Definition 13.4. Given any QBF A, we define the set $FV(A)$ of *variables free in* A and the set $BV(A)$ of *variables bound in* A as follows:

$$\begin{aligned}
FV(\perp) &= FV(\top) = \emptyset \\
FV(x_i) &= \{x_i\} \\
FV(\neg B) &= FV(B) \\
FV((B * C)) &= FV(B) \cup FV(C), \qquad * \in \{\vee, \wedge, \Rightarrow\} \\
FV(\forall x B) &= FV(B) - \{x\} \\
FV(\exists x B) &= FV(B) - \{x\},
\end{aligned}$$

and

$$\begin{aligned}
BV(\perp) &= BV(\top) = \emptyset \\
BV(x_i) &= \emptyset \\
BV(\neg B) &= BV(B) \\
BV((B * C)) &= BV(B) \cup BV(C), \qquad * \in \{\vee, \wedge, \Rightarrow\} \\
BV(\forall x B) &= BV(B) \cup \{x\} \\
BV(\exists x B) &= BV(B) \cup \{x\}.
\end{aligned}$$

A QBF A such that $FV(A) = \emptyset$ (A has no free variables) is said to be *closed* or a *sentence*.

It should be noted that $FV(A)$ and $BV(A)$ may not be disjoint! For example, if

$$A = x_1 \vee \forall x_1(\neg x_1 \vee x_2),$$

then $FV(A) = \{x_1, x_2\}$ and $BV(A) = \{x_1\}$. This situation is somewhat undesirable. Intuitively, A is "equivalent" to the QBF

$$A' = x_1 \vee \forall x_3(\neg x_3 \vee x_2),$$

with $FV(A') = \{x_1, x_2\}$ and $BV(A') = \{x_3\}$. Here equivalent means that A and A' have the same truth value for all truth assignments. To make all this precise we proceed as follows.

Definition 13.5. A *substitution* is a set of pairs $\varphi = \{(y_1, A_1), \ldots, (y_m, A_m)\}$ where the variables $y_1, \ldots, y_m$ are *distinct* and $A_1, \ldots, A_m$ are arbitrary QBF's. We write $\varphi = [y_1 := A_1, \ldots, y_m := A_m]$. For any QBF B, we also denote by $\varphi[y_i := B]$ the substitution such that $y_i := A_i$ is replaced by $y_i := B$. In particular, $\varphi[y_i := y_i]$ leaves y_i unchanged.

Given a QBF A, the result of applying the substitution $\varphi = [y_1 := A_1, \ldots, y_m := A_m]$ to A, denoted $A[\varphi]$, is defined inductively as follows:

$$\begin{aligned}
\bot[\varphi] &= \bot \\
\top[\varphi] &= \top \\
x[\varphi] &= A_i \quad \text{if } x = y_i,\ 1 \leq i \leq m \\
x[\varphi] &= x \quad \text{if } x \notin \{y_1, \ldots, y_m\} \\
(\neg B)[\varphi] &= (\neg B)[\varphi] \\
(B * C)[\varphi] &= (B[\varphi] * C[\varphi]), \qquad * \in \{\vee, \wedge, \Rightarrow\} \\
(\forall x B)[\varphi] &= \forall x B[\varphi[y_i := y_i]] \quad \text{if } x = y_i,\ 1 \leq i \leq m \\
(\forall x B)[\varphi] &= \forall x B[\varphi] \quad \text{if } x \notin \{y_1, \ldots, y_m\} \\
(\exists x B)[\varphi] &= \exists x B[\varphi[y_i := y_i]] \quad \text{if } x = y_i,\ 1 \leq i \leq m \\
(\exists x B)[\varphi] &= \exists x B[\varphi] \quad \text{if } x \notin \{y_1, \ldots, y_m\}.
\end{aligned}$$

Definition 13.6. A QBF A is *rectified* if distinct quantifiers bind distinct variables and if $BV(A) \cap FV(A) = \emptyset$.

Given a QBF A and any finite set V of variables, we can define recursively a new rectified QBF A' such that $BV(A') \cap V = \emptyset$.

(1) If $A = \top$, or $A = \bot$, or $A = x_i$, then $A' = A$.

(2) If $A = \neg B$, then $A' = A$.

(3) If $A = (B \vee C)$, then first we find recursively some rectified QBF B_1 such that $BV(B_1) \cap V = \emptyset$, then we find recursively some rectified QBF C_1 such that $BV(C_1) \cap (FV(B_1) \cup BV(B_1) \cup V) = \emptyset$, and we set $A' = (B_1 \vee C_1)$. We proceed similarly if $A = (B \wedge C)$ or $A = (B \Rightarrow C)$, with $\vee$ replaced by $\wedge$ or $\Rightarrow$.

(4) If $A = \forall x B$, first we find recursively some rectified QBF B_1 such that $BV(B_1) \cap V = \emptyset$, and then we let $A' = \forall z B_1[x := z]$ for some new variable z such that $z \notin FV(B_1) \cup BV(B_1) \cup V$. Note that in this step it is possible that $x \notin FV(B)$.

(5) If $A = \exists x B$, first we find recursively some rectified QBF B_1 such that $BV(B_1) \cap V = \emptyset$, and then we let $A' = \exists z B_1[x := z]$ for some new variable z such that $z \notin FV(B_1) \cup BV(B_1) \cup V$. Note that in this step it is possible that $x \notin FV(B)$.

Given any QBF A, we find a rectified QBF A' by applying the above procedure recursively starting with A and $V = \emptyset$.

Recall that a *truth assignment* or *valuation* is a function $v\colon \mathbf{PV} \to \{\mathbf{T}, \mathbf{F}\}$. We also let $\overline{\mathbf{T}} = \mathbf{F}$ and $\overline{\mathbf{T}} = \mathbf{T}$.

Definition 13.7. Given a valuation $v\colon \mathbf{PV} \to \{\mathbf{T}, \mathbf{F}\}$, we define *truth value* $A[v]$ of a QBF A inductively as follows.

$$
\begin{aligned}
\perp[v] &= \mathbf{F} &(1)\\
\top[v] &= \mathbf{T} &(2)\\
x[v] &= v(x) &(3)\\
(\neg B)[v] &= \overline{B[v]} = \mathbf{F} \text{ if } B[v] = \mathbf{T} \text{ else } \mathbf{T} \text{ if } B[v] = \mathbf{F} &(4)\\
(B \vee C)[v] &= B[v] \;\mathbf{or}\; C[v] &(5)\\
(B \wedge C)[v] &= B[v] \;\mathbf{and}\; C[v] &(6)\\
(B \Rightarrow C)[v] &= \overline{B[v]} \;\mathbf{or}\; C[v] &(7)\\
(\forall x B)[v] &= B[v[x := \mathbf{T}]] \;\mathbf{and}\; B[v[x := \mathbf{F}]] &(8)\\
(\exists x B)[v] &= B[v[x := \mathbf{T}]] \;\mathbf{or}\; B[v[x := \mathbf{F}]]. &(9)
\end{aligned}
$$

If $A[v] = \mathbf{T}$, we write say that v *satisfies* A and we write $v \models A$. If $A[v] = \mathbf{T}$ for *all valuations* v , we say that A *is valid* and we write $\models A$.

As usual, we write $A \equiv B$ iff $(A \Rightarrow B) \wedge (B \Rightarrow A)$ is valid.

In Clause (5) when evaluating $(B \vee C)[v]$, if $B[v] = \mathbf{T}$, then we don't need to evaluate $C[v]$, since $\mathbf{T}\,\mathbf{or}\,b = \mathbf{T}$ independently of $b \in \{\mathbf{T}, \mathbf{F}\}$, and so $(B \vee C)[v] = \mathbf{T}$. If $B[v] = \mathbf{F}$, then we need to evaluate $C[v]$, and $(B \vee C)[v] = \mathbf{T}$ iff $C[v] = \mathbf{T}$. Even though the above method is more economical, we usually evaluate both $B[v]$ and $C[v]$ and then compute $B[v] \;\mathbf{or}\; C[v]$.

A similar discussion applies to evaluating $(\exists x B)[v]$ in Clause (9). If $B[v[x := \mathbf{T}]] = \mathbf{T}$, then we don't need to evaluate $B[v[x := \mathbf{F}]]$ and $(\exists x B)[v] = \mathbf{T}$. If $B[v[x := \mathbf{T}]] = \mathbf{F}$, then we need to evaluate $B[v[x := \mathbf{F}]]$, and $(\exists x B)[v] = \mathbf{T}$ iff $B[v[x := \mathbf{F}]] = \mathbf{T}$. Even though the above method is more economical, we usually evaluate both $B[v[x := \mathbf{T}]]$ and $B[v[x := \mathbf{F}]]$ and then compute $B[v[x := \mathbf{T}]] \;\mathbf{or}\; B[v[x := \mathbf{F}]]$.

Example 13.2. Let us show that the QBF

$$A = \forall x\big(\exists y(x \wedge y) \vee \forall z(\neg x \vee z)\big)$$

from Example 13.1 is valid. This is a closed formula so v is irrelevant. By Clause (8) of Definition 13.7, we need to evaluate $A[x := \mathbf{T}]$ and $A[x := \mathbf{F}]$.

To evaluate $A[x := \mathbf{T}]$, by Clause (5) of Definition 13.7, we need to evaluate $(\exists y(x \wedge y))[x := \mathbf{T}]$ and $(\forall z(\neg x \vee z))[x := \mathbf{T}]$.

To evaluate $(\exists y(x \wedge y))[x := \mathbf{T}]$, by Clause (9) of Definition 13.7, we need to evaluate $(x \wedge y)[x := \mathbf{T}, y := \mathbf{T}]$ and $(x \wedge y)[x := \mathbf{T}, y := \mathbf{F}]$.

We have (by Clause (6)) $(x \wedge y)[x := \mathbf{T}, y := \mathbf{T}] = \mathbf{T}\,\mathbf{and}\,\mathbf{T} = \mathbf{T}$ and $(x \wedge y)[x := \mathbf{T}, y := \mathbf{F}] = \mathbf{T}\ \mathbf{and}\ \mathbf{F} = \mathbf{F}$, so

$$(\exists y(x \wedge y))[x := \mathbf{T}] = (x \wedge y)[x := \mathbf{T}, y := \mathbf{T}]\ \mathbf{or}\ (x \wedge y)[x := \mathbf{T}, y := \mathbf{F}] = \mathbf{T}\ \mathbf{or}\ \mathbf{F} = \mathbf{T}. \quad (1)$$

To evaluate $(\forall z(\neg x \vee z))[x := \mathbf{T}]$, by Clause (8) of Definition 13.7, we need to evaluate $(\neg x \vee z)[x := \mathbf{T}, z := \mathbf{T}]$ and $(\neg x \vee z)[x := \mathbf{T}, z := \mathbf{F}]$.

Using Clauses (4) and (5) of Definition 13.7, we have $(\neg x \vee z)[x := \mathbf{T}, z := \mathbf{T}] = \overline{\mathbf{T}}\ \mathbf{land}\ \mathbf{T} = \mathbf{T}$ and $(\neg x \vee z)[x := \mathbf{T}, z := \mathbf{F}] = \overline{\mathbf{T}}\ \mathbf{land}\ \mathbf{F} = \mathbf{F}$, so

$$(\forall z(\neg x \vee z))[x := \mathbf{T}] = (\neg x \vee z)[x := \mathbf{T}, z := \mathbf{T}]\ \mathbf{and}\ (\neg x \vee z)[x := \mathbf{T}, z := \mathbf{F}] = \mathbf{F}. \quad (2)$$

By (1) and (2) we have

$$A[x := \mathbf{T}] = (\exists y(x \wedge y))[x := \mathbf{T}]\ \mathbf{or}\ (\forall z(\neg x \vee z))[x := \mathbf{T}] = \mathbf{T}\ \mathbf{or}\ \mathbf{F} = \mathbf{T}. \quad (3)$$

Now we need to evaluate $A[x := \mathbf{F}]$. By Clause (5) of Definition 13.7, we need to evaluate $(\exists y(x \wedge y))[x := \mathbf{F}]$ and $(\forall z(\neg x \vee z))[x := \mathbf{F}]$.

To evaluate $(\exists y(x \wedge y))[x := \mathbf{F}]$, by Clause (9) of Definition 13.7, we need to evaluate $(x \wedge y)[x := \mathbf{F}, y := \mathbf{T}]$ and $(x \wedge y)[x := \mathbf{F}, y := \mathbf{F}]$.

We have (by Clause (6)) $(x \wedge y)[x := \mathbf{F}, y := \mathbf{T}] = \mathbf{F}\ \mathbf{and}\ \mathbf{T} = \mathbf{F}$ and $(x \wedge y)[x := \mathbf{F}, y := \mathbf{F}] = \mathbf{F}\ \mathbf{and}\ \mathbf{F} = \mathbf{F}$, so

$$(\exists y(x \wedge y))[x := \mathbf{F}] = (x \wedge y)[x := \mathbf{F}, y := \mathbf{T}]\ \mathbf{or}\ (x \wedge y)[x := \mathbf{F}, y := \mathbf{F}] = \mathbf{F}\ \mathbf{or}\ \mathbf{F} = \mathbf{F}. \quad (4)$$

To evaluate $(\forall z(\neg x \vee z))[x := \mathbf{F}]$, by Clause (8) of Definition 13.7, we need to evaluate $(\neg x \vee z)[x := \mathbf{F}, z := \mathbf{T}]$ and $(\neg x \vee z)[x := \mathbf{F}, z := \mathbf{F}]$.

Using Clauses (4) and (5) of Definition 13.7, we have $(\neg x \vee z)[x := \mathbf{F}, z := \mathbf{T}] = \overline{\mathbf{F}}\,\mathbf{or}\,\mathbf{T} = \mathbf{T}$ and $(\neg x \vee z)[x := \mathbf{F}, z := \mathbf{F}] = \overline{\mathbf{F}}\ \mathbf{or}\ \mathbf{F} = \mathbf{T}$, so

$$(\forall z(\neg x \vee z))[x := \mathbf{F}] = (\neg x \vee z)[x := \mathbf{F}, z := \mathbf{T}]\ \mathbf{and}\ (\neg x \vee z)[x := \mathbf{F}, z := \mathbf{F}] = \mathbf{T} \quad (5)$$

By (4) and (5) we have

$$A[x := \mathbf{F}] = (\exists y(x \wedge y))[x := \mathbf{F}] \ \mathbf{or} \ (\forall z(\neg x \vee z))[x := \mathbf{F}] = \mathbf{F} \ \mathbf{or} \ \mathbf{T} = \mathbf{T}. \tag{6}$$

Finally, by (3) and (6) we get

$$A[x := \mathbf{T}] \ \mathbf{and} \ A[x := \mathbf{F}] = \mathbf{T} \ \mathbf{and} \ \mathbf{T} = \mathbf{T}, \tag{7}$$

so A is valid.

The reader should observe that in evaluating

$$(\exists x B)[v] = B[v[x := \mathbf{T}]] \ \mathbf{or} \ B[v[x := \mathbf{F}]],$$

if $(\exists x B)[v] = \mathbf{T}$, it is only necessary to guess which of $B[v[x := \mathbf{T}]]$ or $B[v[x := \mathbf{F}]]$ evaluates to $\mathbf{T}$, so we can view the computation of $A[v]$ as an AND/OR tree, where an AND node corresponds to the evaluation of a formula $(\forall x B)[v]$, and an OR node corresponds to the evaluation of a formula $(\exists x B)[v]$.

Evaluating the truth value $A[v]$ of a QBF A can take exponential time in the size n of A, but we will see that it only requires $O(n^2)$ space. Also, the validity of QBF's of the form

$$\exists x_1 \exists x_2 \cdots \exists x_m B$$

where B is quantifier-free and $FV(B) = \{x_1, \ldots, x_m\}$ is equivalent to SAT (the satisfiability problem), and the validity of QBF's of the form

$$\forall x_1 \forall x_2 \cdots \forall x_m B$$

where B is quantifier-free and $FV(B) = \{x_1, \ldots, x_m\}$ is equivalent to TAUT (the validity problem). This is why the validity problem for QBF's is as hard as both SAT and TAUT.

We mention the following technical results. Part (1) and Part (2) are used all the time.

Proposition 13.5. *Let A be any QBF.*

(1) For any two valuations v_1 and v_2, if $v_1(x) = v_2(x)$ for all $x \in FV(A)$, then $A[v_1] = A[v_2]$. In particular, if A is a sentence, then $A[v]$ is independent of v.

(2) If A' is any rectified QBF obtained from A, then $A[v] = A'[v]$ for all valuations v; that is, $A \equiv A'$.

(3) For any QBF A of the form $A = \forall x B$ and any QBF C such that $BV(B) \cap FV(C) = \emptyset$, if A is valid, then $B[x := C]$ is also valid.

(4) For any QBF B and any QBF C such that $BV(B) \cap FV(C) = \emptyset$, if $B[x := C]$ is valid, then $\exists x B$ is also valid.

We also repeat Proposition 6.13 which states that the connectives $\wedge, \vee, \neg$ and $\exists$ are definable in terms of $\Rightarrow$ and $\forall$. This shows the power of the second-order quantifier $\forall$.

Proposition 13.6. *The connectives $\wedge, \vee, \neg, \perp$ and $\exists$ are definable in terms of $\Rightarrow$ and $\forall$, which means that the following equivalences are valid, where x is not free in B or C:*

$$\begin{aligned} B \wedge C &\equiv \forall x\big((B \Rightarrow (C \Rightarrow x)) \Rightarrow x\big) \\ B \vee C &\equiv \forall x\big((B \Rightarrow x) \Rightarrow ((C \Rightarrow x) \Rightarrow x)\big) \\ \perp &\equiv \forall x x \\ \neg B &\equiv B \Rightarrow \forall x x \\ \exists y B &\equiv \forall x\big((\forall y(B \Rightarrow x)) \Rightarrow x\big). \end{aligned}$$

We now prove the first step in establishing that the validity problem for QBF's is $\mathcal{PS}$-complete.

Proposition 13.7. *Let A be any QBF of length n. Then for any valuation v, the truth value $A[v]$ can be evaluated in $O(n^2)$ space. Thus the validity problem for closed QBF's is in $\mathcal{PS}$.*

Proof. The clauses of Definition 13.7 show that $A[v]$ is evaluated recursively. In clauses (5)-(9), even though two recursive calls are performed, it is only necessary to save one of the two stack frames at a time. It follows that the stack will never contain more than n stack frames, and each stack frame has size at most n. Thus only $O(n^2)$ space is needed. For more details, see Hopcroft, Motwani and Ullman [29] (Section 11.3.4, Theorem 11.10). $\square$

Finally we state the main theorem proven by Meyer and Stockmeyer (1973).

Theorem 13.8. *The validity problem for closed QBF's is $\mathcal{PS}$-complete.*

We will not prove Theorem 13.8, mostly because it requires simulating the computation of a polynomial-space bounded deterministic Turing machine, and this is very technical and tedious. Most details of such a proof can be found in Hopcroft, Motwani and Ullman [29] (Section 11.3.4, Theorem 11.11).

Let us simply make the following comment which gives a clue as to why QBF's are helpful in describing the simulation (for details, see Hopcroft, Motwani and Ullman [29] (Theorem 11.11)). It turns out that the idea behind the function *reach* presented in Section 13.2 plays a key role. It is necessary to express for any two ID's I and J and any $i \geq 1$, that $I \vdash^k J$ with $k \leq i$. This is achieved by defining $N_{2i}(I, J)$ as the following QBF:

$$N_{2i}(I, J) = \exists K \forall R \forall S\Big(\big((R = I \wedge S = K) \vee (R = K \wedge S = J)\big) \Rightarrow N_i(R, S)\Big).$$

Another interesting $\mathcal{PS}$-complete problem due to Karp (1972) is the following. Given any alphabet Σ, decide whether a regular expression R denotes Σ^*; that is, $\mathcal{L}[R] = \Sigma^*$.

We conclude with some comments regarding some remarkable results of Statman regarding the connection between validity of closed QBF's and provability in intuitionistic propositional logic.

13.4 Complexity of Provability in Intuitionistic Propositional Logic

Recall that intuitionistic logic is obtained from classical logic by taking away the proof-by-contradiction rule. The reader is strongly advised to review Chapter 2, especially Sections 2.2, 2.3, 2.4, 2.6 and 2.7, before proceeding.

Statman [46] shows how to reduce the validity problem for QBFs to provability in intuitionistic propositional logic. To simplify the construction we may assume that we consider QBF's in *prenex form*, which means that they are of the form

$$A = Q_n x_n Q_{n-1} x_{n-1} \cdots Q_1 x_1 B_0$$

where B_0 is quantifier-free and $Q_i \in \{\forall, \exists\}$ for $i = 1, \ldots, n$. We also assume that A is rectified. It is easy to show that any QBF A is equivalent to some QBF A' in prenex form by adapting the method for converting a first-order formula to prenex form; see Gallier [19] or Shoenfield [45].

Statman's clever trick is to exploit some properties of intuitionistic provability that do not hold for classical logic. One of these properties is that if a proposition $B \vee C$ is provable intuitionistically, we write $\vdash_I B \vee C$, then either $\vdash_I B$ or $\vdash_I C$, that is, either B is provable or C is provable (of course, intuitionistically). This fact is used in the "easy direction" of the proof of Theorem 13.9.

To illustrate the power of the above fact, in his construction, Statman associates the proposition

$$(x \Rightarrow B) \vee (\neg x \Rightarrow B) \tag{$*$}$$

to the QBF $\exists x B$. Classically this is useless, because $(*)$ is classically valid, but if $(*)$ is *intuitionistically provable*, then either $x \Rightarrow B$ is provable or $\neg x \Rightarrow B$ is intuitionistically provable, but this implies that either $x \Rightarrow B$ is classically provable or $\neg x \Rightarrow B$ is classically provable, and so either $B[x := \mathbf{T}]$ is valid or $B[x := \mathbf{F}]$ is valid, which means that $\exists x B$ is valid.

As a first step, Statman defines the proposition B_k^+ inductively as follows: for all k such that $0 \leq k \leq n-1$,

$$\begin{aligned}
B_0^+ &= \neg\neg B_0 & & \\
B_{k+1}^+ &= (x_{k+1} \vee \neg x_{k+1}) \Rightarrow B_k^+ & & \text{if } Q_{k+1} = \forall \\
B_{k+1}^+ &= (x_{k+1} \Rightarrow B_k^+) \vee (\neg x_{k+1} \Rightarrow B_k^+), & & \text{if } Q_{k+1} = \exists
\end{aligned}$$

and set $A^+ = B_n^+$. Obviously A^+ is quantifier-free. We also let $B_{k+1} = Q_{k+1} x_{k+1} B_k$ for $k = 0, \ldots n-1$, so that $A = B_n$.

The following example illustrates the above definition.

Example 13.3. Consider the QBF is prenex form

$$A = \exists x_3 \forall x_2 \exists x_1((x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2) \wedge (\neg x_3 \vee x_1)).$$

It is indeed valid, as we see by setting $x_3 = \mathbf{F}$, and if $x_2 = \mathbf{T}$ then $x_1 = \mathbf{F}$, else if $x_2 = \mathbf{F}$ then $x_1 = \mathbf{T}$. We have

$$\begin{aligned}
B_0^+ &= \neg\neg((x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2) \wedge (\neg x_3 \vee x_1)) \\
B_1^+ &= (x_1 \Rightarrow B_0^+) \vee (\neg x_1 \Rightarrow B_0^+) \\
B_2^+ &= (x_2 \vee \neg x_2) \Rightarrow B_1^+ \\
B_3^+ &= (x_3 \Rightarrow B_2^+) \vee (\neg x_3 \Rightarrow B_2^+),
\end{aligned}$$

and $A^+ = B_3^+$.

Statman proves the following remarkable result (Statman [46], Proposition 1).

Theorem 13.9. *For any closed QBF A in prenex form, A is valid iff $\vdash_I A^+$; that is, A^+ is intuitionistically provable.*

Proof sketch. Here is a sketch of Statman's proof using the QBF of Example 13.3. First assume the QBF A is valid. The first step is to eliminate existential quantifiers using a variant of what is known as Skolem functions; see Gallier [19] or Shoenfield [45].

The process is to assign to the jth existential quantifier $\exists x_k$ from the left in the formula $Q_n x_n \cdots Q_1 x_1 B_0$ a boolean function C_j depending on the universal quantifiers $\forall x_{i_1}, \ldots, \forall x_{i_p}$ to the left of $\exists x_k$ and defined such that $Q_n x_n \cdots Q_{k+1} x_{k+1} \exists x_k B_{k-1}$ is valid iff $\forall x_{i_1} \cdots \forall x_{i_q} B_{k-1}^s$ is valid, where B_{k-1}^s is the result of substituting the functions $C_1, \ldots, C_j$ associated with the j existential quantifiers from the left for these existentially quantified variables.

We associate with $\exists x_3$ the constant C_1 such that $C_1 = \mathbf{F}$, and with $\exists x_1$ the boolean function $C_2(x_2)$ given by

$$C_2(\mathbf{T}) = \mathbf{F}, \quad C_2(\mathbf{F}) = \mathbf{T}.$$

The constant C_1 and the function C_2 are chosen so that

$$A = \exists x_3 \forall x_2 \exists x_1((x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2) \wedge (\neg x_3 \vee x_1))$$

is valid iff

$$A^s = \forall x_2((C_2(x_2) \vee x_2) \wedge (\neg C_2(x_2) \vee \neg x_2) \wedge (\neg C_1 \vee C_2(x_2))) \tag{S}$$

is valid. Indeed, since $C_1 = \mathbf{F}$, the clause $(\neg C_1 \vee C_2(x_2))$ evaluates to $\mathbf{T}$ regardless of the value of x_2, and by definition of C_2, the expression

$$\forall x_2(C_2(x_2) \vee x_2) \wedge (\neg C_2(x_2) \vee \neg x_2))$$

also evaluates to $\mathbf{T}$. We now build a tree of Gentzen sequents (from the root up) from the expression in (S) which guides us in deciding which disjunct to pick when dealing with a proposition B_k^+ associated with an existential quantifier. Here is the tree.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\neg x_3, x_2, \neg x_1 \to B_0^+}{\neg x_3, x_2 \to \neg x_1 \Rightarrow B_0^+}}{\neg x_3, x_2 \to (x_1 \Rightarrow B_0^+) \vee (\neg x_1 \Rightarrow B_0^+)} \qquad \dfrac{\dfrac{\neg x_3, \neg x_2, x_1 \to B_0^+}{\neg x_3, \neg x_2 \to x_1 \Rightarrow B_0^+}}{\neg x_3, \neg x_2 \to (x_1 \Rightarrow B_0^+) \vee (\neg x_1 \Rightarrow B_0^+)}}{\neg x_3, x_2 \vee \neg x_2 \to B_1^+}}{\neg x_3 \to (x_2 \vee \neg x_2) \Rightarrow B_1^+}}{\to \neg x_3 \Rightarrow B_2^+}}{\to (x_3 \Rightarrow B_2^+) \vee (\neg x_3 \Rightarrow B_2^+)}$$

We will see that by adding subtrees proving the sequents in the leaf nodes, this tree becomes an intuitionistic proof of A^+. Note that this a proof in a Gentzen sequent style formulation of intuitionistic logic (see Kleene [32], Gallier [14], Takeuti [49]), not a proof in a natural deduction style proof system as in Section 2.6.

The tree is constructed from the bottom-up starting with $\to A^+$. For every leaf node in the tree where a sequent is of the form

$$\ell_n, \ldots, \ell_{k+1} \to (x_k \Rightarrow B_{k-1}^+) \vee (\neg x_k \Rightarrow B_{k-1}^+)$$

where $\ell_n, \ldots, \ell_{k+1}$ are literals, we know that $Q_k = \exists$ is the jth existential quantifier from the left, so we use the boolean function C_j to determine which of the two disjuncts $x_k \Rightarrow B_{k-1}^+$ or $\neg x_k \Rightarrow B_{k-1}^+$ to keep. The function C_j depends on the value of the literals $\ell_n, \ldots, \ell_{k+1}$ associated with universal quantifiers (where ℓ_i has the value $\mathbf{T}$ if $\ell_i = x_i$ and ℓ_i has the value $\mathbf{F}$ if $\ell_i = \neg x_i$). Even though C_j is independent of the value of the literals ℓ_i associated with existential quantifiers, to simplify notation we write $C_j(\ell_n, \ldots, \ell_{k+1})$ for the value of the function C_j. If $C_j(\ell_n, \ldots, \ell_{k+1}) = \mathbf{T}$, then we pick the disjunct $x_k \Rightarrow B_{k-1}^+$, else if $C_j(\ell_n, \ldots, \ell_{k+1}) = \mathbf{F}$, then we pick the disjunct $\neg x_k \Rightarrow B_{k-1}^+$. Denote the literal corresponding to the chosen disjunct by ℓ_k ($\ell_k = x_k$ in the first case, $\ell_k = \neg x_k$ in the second case). Then we grow two new nodes

$$\ell_n, \ldots, \ell_{k+1} \to \ell_k \Rightarrow B_{k-1}^+$$

and

$$\ell_n, \ldots, \ell_{k+1}, \ell_k \to B_{k-1}^+$$

above the (leaf) node

$$\ell_n, \ldots, \ell_{k+1} \to (x_k \Rightarrow B_{k-1}^+) \vee (\neg x_k \Rightarrow B_{k-1}^+).$$

For every leaf node of the form

$$\ell_n, \ldots, \ell_{k+1} \to (x_k \vee \neg x_k) \Rightarrow B_{k-1}^+,$$

we grow the new node

$$\ell_n, \ldots, \ell_{k+1}, x_k \vee \neg x_k \to B_{k-1}^+,$$

and then the two new nodes (both descendants of the above node, so there is branching in the tree),

$$\ell_n, \ldots, \ell_{k+1}, x_k \to B_{k-1}^+ \quad \text{and} \quad \ell_n, \ldots, \ell_{k+1}, \neg x_k \to B_{k-1}^+.$$

By induction from the bottom-up, since A is valid and since the tree was constructed in terms of the constant C_1 and the function C_2 which ensure the validity of A, it is easy to see that for every node $\ell_n, \ldots, \ell_{k+1} \to B_k^+$, the sequent $\ell_n, \ldots, \ell_{k+1} \to B_k$ (note, the right-hand side is the original formula B_k) is classically valid, and thus classically provable (by the completeness theorem for propositional logic). Consequently *every leaf* $\ell_n, \ldots, \ell_1 \to B_0$ is classically provable, so by Glivenko's theorem (see Kleene [32] (Theorem 59), or Gallier [14] (Section 13)), the sequent $\ell_n, \ldots, \ell_1 \to \neg\neg B_0$ is *intuitionistically provable*. But this is the sequent $\ell_n, \ldots, \ell_1 \to B_0^+$ so all the leaves of the tree are intuitionistically provable, and since the tree is a deduction tree in a Gentzen sequent style formulation of intuitionistic logic (see Kleene [32], Gallier [14], Takeuti [49]), the root $A^+ = B_n^+$ is intuitionistically provable.

In the other direction, assume that A^+ is intuitionistically provable. We use the fact that if

$$\ell_n, \ldots, \ell_k \to A \vee B$$

is intuitionistically provable and the ℓ_i are literals, then either $\ell_n, \ldots, \ell_k \to A$ is intuitionistically provable or $\ell_n, \ldots, \ell_k \to B$ is intuitionistically provable, and other proof rules of intuitionistic logic (see Kleene [32], Gallier [14], Takeuti [49]), to build a proof tree just as we did before. Then every sequent $\ell_n, \ldots, \ell_{k+1} \to B_k^+$ is intuitionistically provable, thus classically provable, and consequently classically valid. But this immediately implies (by induction starting from the leaves) that $\ell_n, \ldots, \ell_{k+1} \to B_k$ is also classically valid for all k, and thus $A = B_n$ is valid. $\square$

Statman does not specifically state which proof system of intuitionistic logic is used in Theorem 13.9. Careful inspection of the proof shows that we can construct proof trees in a Gentzen sequent calculus as described in Gallier [14] (system $\mathcal{G}_i$, Section 4) or Kleene [32] (system G3a, Section 80, pages 481-482). This brings up the following issue: could we use instead proofs in natural deduction style, as in Prawitz [41] or Gallier [14]? The answer is yes, because there is a polynomial-time translation of intuitionistic proofs in Gentzen sequent style to intuitionistic proofs in natural deduction style, as shown in Gallier [14], Section 5. So Theorem 13.9 applies to a Gentzen sequent style proof system or to a natural deduction style proof system.

The problem with the translation $A \mapsto A^+$ is that A^+ may not have size polynomial in the size (the length of A as a string) of A because in the case of an existential quantifier the length of the formula B_{k+1}^+ is more than twice the length of the formula B_k^+, so Statman introduces a second translation.

The proposition $B_k^\dagger$ is defined inductively as follows. Let $y_0, y_1, \ldots, y_n$ be $n + 1$ new propositional variables. For all k such that $0 \leq k \leq n-1$,

$$\begin{aligned}
B_0^{\dagger} &= \neg\neg B_0 \equiv y_0 && \\
B_{k+1}^{\dagger} &= ((x_{k+1} \vee \neg x_{k+1}) \Rightarrow y_k) \equiv y_{k+1} && \text{if } Q_{k+1} = \forall \\
B_{k+1}^{\dagger} &= ((x_{k+1} \Rightarrow y_k) \vee (\neg x_{k+1} \Rightarrow y_k)) \equiv y_{k+1}, && \text{if } Q_{k+1} = \exists
\end{aligned}$$

and set

$$A^* = B_0^{\dagger} \Rightarrow (B_1^{\dagger} \Rightarrow (\cdots (B_n^{\dagger} \Rightarrow y_n) \cdots)).$$

It is easy to see that the translation $A \mapsto A^*$ can be done in polynomial space. Statman proves the following result (Statman [46], Proposition 2).

Theorem 13.10. *For any closed QBF A in prenex form, $\vdash_I A^+$ iff $\vdash_I A^*$; that is, A^+ is intuitionistically provable iff A^* is intuitionistically provable.*

Proof. First suppose the sequent $\to A^+$ is provable (in Kleene G3a). We claim that the sequent

$$B_0^{\dagger}, \ldots, B_k^{\dagger} \to B_k^+ \equiv y_k$$

is provable for $k = 0, \ldots, n$. We proceed by induction on k. For the base case $k = 0$, we have $B_0^{\dagger} = (\neg\neg B_0 \equiv y_0)$ and $B_0^+ = \neg\neg B_0$, so $B_0^{\dagger} \to (B_0^+ \equiv y_0) = (B_0^+ \equiv y_0) \to (B_0^+ \equiv y_0)$, which is an axiom.

For the induction step, if $Q_{k+1} = \forall$, then

$$B_{k+1}^+ = (x_{k+1} \vee \neg x_{k+1}) \Rightarrow B_k^+, \qquad B_{k+1}^{\dagger} = ((x_{k+1} \vee \neg x_{k+1}) \Rightarrow y_k) \equiv y_{k+1},$$

by the induction hypothesis

$$B_0^{\dagger}, \ldots, B_k^{\dagger} \to B_k^+ \equiv y_k$$

is provable, and since the sequent

$$B_0^{\dagger}, \ldots, B_k^{\dagger}, B_{k+1}^{\dagger} \to B_{k+1}^{\dagger}$$

is an axiom, by substituting B_k^+ for y_k in $B_{k+1}^{\dagger} = ((x_{k+1} \vee \neg x_{k+1}) \Rightarrow y_k) \equiv y_{k+1}$ in the conclusion of the above sequent, we deduce that

$$B_0^{\dagger}, \ldots, B_k^{\dagger}, B_{k+1}^{\dagger} \to ((x_{k+1} \vee \neg x_{k+1}) \Rightarrow B_k^+) \equiv y_{k+1}$$

is provable. Since $B_{k+1}^+ = (x_{k+1} \vee \neg x_{k+1}) \Rightarrow B_k^+$, we conclude that

$$B_0^{\dagger}, \ldots, B_k^{\dagger}, B_{k+1}^{\dagger} \to B_{k+1}^+ \equiv y_{k+1}$$

is provable.

If $Q_{k+1} = \exists$, then

$$B_{k+1}^+ = (x_{k+1} \Rightarrow B_k^+) \vee (\neg x_{k+1} \Rightarrow B_k^+), \qquad B_{k+1}^{\dagger} = ((x_{k+1} \Rightarrow y_k) \vee (x_{k+1} \Rightarrow y_k)) \equiv y_{k+1},$$

by the induction hypothesis

$$B_0^\dagger, \ldots, B_k^\dagger \rightarrow B_k^+ \equiv y_k$$

is provable, and since the sequent

$$B_0^\dagger, \ldots, B_k^\dagger, B_{k+1}^\dagger \rightarrow B_{k+1}^\dagger$$

is an axiom, by substituting B_k^+ for y_k in $B_{k+1}^\dagger = ((x_{k+1} \Rightarrow y_k) \vee (x_{k+1} \Rightarrow y_k)) \equiv y_{k+1}$ in the conclusion of the above sequent, we deduce that

$$B_0^\dagger, \ldots, B_k^\dagger, B_{k+1}^\dagger \rightarrow ((x_{k+1} \Rightarrow B_k^+) \vee (\neg x_{k+1} \Rightarrow B_k^+)) \equiv y_{k+1}$$

is provable. Since $B_{k+1}^+ = (x_{k+1} \Rightarrow B_k^+) \vee (\neg x_{k+1} \Rightarrow B_k^+)$, we conclude that

$$B_0^\dagger, \ldots, B_k^\dagger, B_{k+1}^\dagger \rightarrow B_{k+1}^+ \equiv y_{k+1}$$

is provable. Therefore the induction step holds. For $k = n$, we see that the sequent

$$B_0^\dagger, \ldots, B_n^\dagger \rightarrow (B_n^+ \equiv y_n) = B_0^\dagger, \ldots, B_n^\dagger \rightarrow (A^+ \equiv y_n)$$

is provable, and since by hypothesis $\rightarrow A^+$ is provable, we deduce that

$$B_0^\dagger, \ldots, B_n^\dagger \rightarrow y_n$$

is provable. Finally we deduce that

$$A^* = B_0^\dagger \Rightarrow (B_1^\dagger \Rightarrow (\cdots (B_n^\dagger \Rightarrow y_n) \cdots))$$

is provable intuitionistically.

Conversely assume that $A^* = B_0^\dagger \Rightarrow (B_1^\dagger \Rightarrow (\cdots (B_n^\dagger \Rightarrow y_n) \cdots))$ is provable intuitionistically. Then using basic properties of intuitionistic provability, the sequent

$$B_0^\dagger, \ldots, B_n^\dagger \rightarrow y_n$$

is provable intuitionistically. Now if we substitute B_{k+1}^+ for y_{k+1} in $B_{k+1}^\dagger$ for $k = 0, \ldots, n-1$, we see immediately that

$$B_{k+1}^\dagger[y_{k+1} := B_{k+1}^+] = B_{k+1}^+ \equiv B_{k+1}^+,$$

so the proof of

$$B_0^\dagger, \ldots, B_n^\dagger \rightarrow y_n$$

yields a proof of

$$B_0^+ \equiv B_0^+, \ldots, B_n^+ \equiv B_n^+ \rightarrow B_n^+,$$

that is, a proof (intuitionistic) of $B_n^+ = A^+$.

Remark: Note that we made implicit use of the cut rule several times, but by Gentzen's cut-elimination theorem this does not matter (see Gallier [14]). $\square$

Using Theorems 13.9 and 13.10 we deduce from the fact that validity of QBF's is $\mathcal{PS}$-complete that provability in propositional intuitionistic logic is $\mathcal{PS}$-hard (every problem in $\mathcal{PS}$ reduces in polynomial time to provability in propositional intuitionistic logic). Using results of Tarski and Ladner, it is can be shown that validity in Kripke models for propositional intuitionistic logic belongs to $\mathcal{PS}$, so Statman proves the following result (Statman [46], Section 2, Theorem).

Theorem 13.11. *The problem of deciding whether a proposition is valid in all Kripke models is $\mathcal{PS}$-complete.*

Theorem 13.11 also applies to any proof system for intuitionisic logic which is sound and complete for Kripke semantics.

Theorem 13.12. *The problem of deciding whether a proposition is intuitionistically provable in any sound and complete proof system (for Kripke semantics) is $\mathcal{PS}$-complete.*

Theorem 13.12 applies to Gallier's system $\mathcal{G}_i$, to Kleene's system G3a, and to natural deduction systems. To prove that $\mathcal{G}_i$ is complete for Kripke semantics it is better to convert proofs in $\mathcal{G}_i$ to proofs in a system due to Takeuti, the system denoted $\mathcal{GKT}_i$ in Gallier [14]; see Section 9, Definition 9.3. Since there is a polynomial-time translation of proofs in $\mathcal{G}_i$ to proofs in natural deduction, the latter system is also complete. This is also proven in van Dalen [51].

Statman proves an even stronger remarkable result, namely that $\mathcal{PS}$-completeness holds even for propositions using only the connective $\Rightarrow$ (Statman [46], Section 2, Proposition 3).

Theorem 13.13. *There is an algorithm which given any proposition A constructs another proposition $A^\sharp$ only involving $\perp, \Rightarrow$, such that that $\vdash_I A$ iff $\vdash_I A^\sharp$.*

Theorem 13.13 is somewhat surprising in view of the fact that $\vee, \wedge, \Rightarrow$ are independent connectives in propositional intuitionistic logic. Finally Statman obtains the following beautiful result (Statman [46], Section 2, Theorem).

Theorem 13.14. *The problem of deciding whether a proposition only involving $\perp, \Rightarrow$ is valid in all Kripke models, and intuitionistically provable in any sound and complete proof system, is $\mathcal{PS}$-complete.*

We highly recommend reading Statman [46], but we warn the reader that this requires perseverance.

Bibliography

[1] George E. Andrews. *Number Theory.* Dover, first edition, 1994.

[2] Tom M. Apostol. *Introduction to Analytic Number Theory.* Undergraduate Texts in Mathematics. Springer, first edition, 1976.

[3] Sanjeev Arora and Boaz Barak. *Computational Complexity. A Modern Approach.* Cambridge University Press, first edition, 2009.

[4] H.P. Barendregt. *The Lambda Calculus. Its Syntax and Semantics.* Studies in Logic No. 103. North-Holland, second revised edition, 1984.

[5] H.P. Barendregt. Lambda Calculi With Types. In Abramsky, Gabbay, and Maibaum, editors, *Handbook of Logic in Computer Science*, pages 117–309. OUP, 1992.

[6] Richard Crandall and Carl Pomerance. *Prime Numbers. A Computational Perspective.* Springer, second edition, 2005.

[7] H.B. Curry and R. Feys. *Combinatory Logic, Vol. I.* Studies in Logic. North-Holland, third edition, 1974.

[8] Martin Davis. Hilbert's tenth problem is unsolvable. *American Mathematical Monthly*, 80(3):233–269, 1973.

[9] Martin Davis, Yuri Matijasevich, and Julia Robinson. Hilbert's tenth problem. diophantine equations: Positive aspects of a negative solution. In *Mathematical Developments Arising from Hilbert Problems*, volume XXVIII, Part 2, pages 323–378. AMS, 1976.

[10] Burton Dreben and Warren D. Goldfarb. *The Decision Problem. Solvable Classes of Quantificational Formulas.* Addison Wesley, first edition, 1979.

[11] Herbert B. Enderton. *Elements of Set Theory.* Academic Press, 1997.

[12] Herbert B. Enderton. *A Mathematical Introduction to Logic.* Elsevier, second edition, 2001.

[13] Jean Gallier. What's so special about Kruskal's theorem and the ordinal Γ_0? A survey of some results in proof theory. *Annals of Pure and Applied Logic*, 53:199–260, 1991.

[14] Jean Gallier. Constructive Logics. Part I: A Tutorial on Proof Systems and Typed λ-Calculi. *Theoretical Computer Science*, 110(2):249–339, 1993.

[15] Jean Gallier. On the Correspondence Between Proofs and λ-Terms. In Philippe de Groote, editor, *Cahiers Du Centre de Logique, Vol. 8*, pages 55–138. Academia, Louvain-La-Neuve, 1995.

[16] Jean Gallier and Jocelyn Quaintance. Notes on Primality Testing and Public Key Cryptography. Part I: Randomized Algorithms, Miller–Rabin and Solovay–Strassen Tests. Technical report, University of Pennsylvania, Levine Hall, Philadelphia, PA 19104, 2017. pdf file available from http://www.cis.upenn.edu/∼jean/RSA-primality-testing.pdf.

[17] Jean H. Gallier. On Girard's "candidats de reductibilité". In P. Odifreddi, editor, *Logic And Computer Science*, pages 123–203. Academic Press, London, New York, May 1990.

[18] Jean H. Gallier. *Discrete Mathematics*. Universitext. Springer Verlag, first edition, 2011.

[19] Jean H. Gallier. *Logic For Computer Science; Foundations of Automatic Theorem Proving*. Dover, second edition, 2015.

[20] G. Gentzen. Investigations into logical deduction. In M.E. Szabo, editor, *The Collected Papers of Gerhard Gentzen*. North-Holland, 1969.

[21] J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*, volume 7 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1989.

[22] Jean-Yves Girard. Une extension de l'interprétation de Gödel à l'analyse, et son application à l'élimination des coupures dans l'analyse et la théorie des types. In J.E. Fenstad, editor, *Proc. 2nd Scand. Log. Symp.*, pages 63–92. North-Holland, 1971.

[23] Jean-Yves Girard. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, Université de Paris VII, June 1972. Thèse de Doctorat d'Etat.

[24] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[25] Gödel, Church, Turing, Rosser, Kleene, and Post. Basic papers on undecidable propositions, unsolvable problems, and computable functions. In Martin Davis, editor, *The Undecidable*. Raven Press, New York, 1965.

[26] Timothy Gowers. *Mathematics: A very Short Introduction*. Oxford University Press, first edition, 2002.

[27] Paul R. Halmos. *Measure Theory*. GTM, Vol. No 18. Springer-Verlag, first edition, 1978.

[28] J.R. Hindley and J.P. Seldin. *Introduction to Combinators and λ-Calculus*, volume 1 of *London Mathematical Society Student texts*. Cambridge University Press, 1986.

[29] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Automata Theory, Languages,and Computation*. Addison Wesley, third edition, 2007.

[30] W. A. Howard. The formulae-as-types notion of construction. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 479–490. Academic Press, London, 1980. Reprint of manuscript first published in 1969.

[31] Michael Huth and Mark Ryan. *Logic in Computer Science. Modelling and reasonning about systems*. Cambridge University Press, Cambridge, United Kingdom, 2000.

[32] S. Kleene. *Introduction to Metamathematics*. North-Holland, seventh edition, 1952.

[33] Jean-Louis Krivine. *Lambda-calcul types et modèles*. Masson, first edition, 1990.

[34] Harry R. Lewis. *Unsolvable Classes of Quantificational Formulas*. Addison Wesley, first edition, 1979.

[35] Harry R. Lewis and Christos H. Papadimitriou. *Elements of the Theory of Computation*. Prentice-Hall, second edition, 1997.

[36] Michael Machtey and Paul Young. *An Introduction to the General Theory of Algorithms*. Elsevier North-Holland, first edition, 1978.

[37] Zohar Manna. *Mathematical Theory of Computation*. Dover, first edition, 2003.

[38] Christos H. Papadimitriou. *Computational Complexity*. Pearson, first edition, 1993.

[39] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, first edition, 2002.

[40] Vaughan R. Pratt. Every prime has a succinct certificate. *SIAM Journal on Computing*, 4(3):214–220, 1975.

[41] D. Prawitz. *Natural deduction, a proof-theoretical study*. Almquist & Wiksell, Stockholm, 1965.

[42] D. Prawitz. Ideas and results in proof theory. In J.E. Fenstad, editor, *Proc. 2nd Scand. Log. Symp.*, pages 235–307. North-Holland, 1971.

[43] Paulo Ribenboim. *The Little Book of Bigger Primes*. Springer-Verlag, second edition, 2004.

[44] Hartley Jr. Rogers. *Theory of Recursive Functions and Effective Computability*. MIT Press, first edition, 1987.

[45] Joseph Shoenfield. *Mathematical Logic*. Addison Wesley, first edition, 2001.

[46] Richard Statman. Intuitionistic propositional logic is polynomial-space complete. *Theoretical Computer Science*, 9(1):67–72, 1979.

[47] Richard Statman. The typed λ-calculus is not elementary recursive. *Theoretical Computer Science*, 9(1):73–81, 1979.

[48] Patrick Suppes. *Axiomatic Set Theory*. Dover, 1972.

[49] G. Takeuti. *Proof Theory*, volume 81 of *Studies in Logic*. North-Holland, 1975.

[50] A.S. Troelstra and H. Schwichtenberg. *Basic Proof Theory*, volume 43 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1996.

[51] D. van Dalen. *Logic and Structure*. Universitext. Springer Verlag, second edition, 1980.

Symbol Index

(MN), 107, 247
(W, φ), 101
0, 117
2^A, 54, 137
$=$, 117
$A - B$, 52, 139
$A \cap B$, 51
$A \cup B$, 51, 137
$A \cap B$, 138
$A \subset B$, 51, 137
$A \subseteq B$, 51, 137
A^+, 53, 137, 142
$M[\varphi]$, 111, 251
$M[x := N]$, 111
$P(\tau_1, \ldots, \tau_n)$, 117, 118
$P[\tau/t]$, 119
$P[\tau_1/t_1, \ldots, \tau_n/t_n]$, 45, 118
$P[u/t]$, 119, 120
$P \equiv Q$, 118
$P \Rightarrow Q$, 13, 65, 118
$P \wedge Q$, 13, 65, 118
$P \vee Q$, 13, 65, 118
S, 117
$S^n(0)$, 130
$X \in Y$, 136
$[x_1 := N_1, \ldots, x_n := N_n]$, 110, 250
Δ, 70
$\Gamma \vdash P$, 85
Γ, 69, 70
Γ, P, 68, 69
Γ, P^x, 70
Π, 70
$\bigcap$, 55, 140
$\bigcap X$, 55, 140
$\bigcup X$, 54, 140
$\bigcup$, 54, 140
$\cap$, 51, 138
$\stackrel{*}{\longleftrightarrow}_\beta$, 112, 252
$\cup$, 51, 137
$\emptyset$, 51, 136
$\equiv$, 14, 33, 66, 85
$\equiv_\alpha$, 111
$\exists t P$, 118
$\exists$, 13, 43, 65, 116, 119, 120
$\forall t P$, 118
$\forall$, 13, 43, 65, 116, 119
$\Rightarrow$, 13, 40, 64, 65, 70, 81, 98
$\longrightarrow_\beta$, 111, 252
$\in$, 50, 135, 136, 144
$\mathbb{Z}$, 36
$\Gamma \triangleright M \colon \sigma$, 108
$\lambda x \colon \sigma. M$, 107
$\lambda x. M$, 247
$\wedge$, 13, 40, 65, 82, 98
$\vee$, 13, 40, 65, 82, 98
$\varphi(u) \colon \mathbf{PS} \to \mathbf{BOOL}$, 101
BOOL, 40, 97
CS, 117
FS, 117
L, 117
PS, 16, 69, 117
false, 13, 40, 65, 97
true, 13, 40, 65, 97
pred, 133
$\mathbb{N}$, 36
$\neg P$, 13, 65, 118

$\neg$, 13, 40, 65, 83, 84, 98
$\oplus$, 154
$\overline{A}$, 139
$\perp$, 13, 65, 83, 118
$\mathbb{Q}$, 37
$\mathbb{R}$, 37
$\xrightarrow{*}_{\beta}$, 112, 252
$\xrightarrow{+}_{\beta}$, 112, 252
$P_1, \ldots, P_m \to Q_1, \ldots, Q_n$, 157
$\Gamma \to P$, 79, 84
$\Gamma \to \Delta$, 157
$\sqrt{2}$, 38
$\subset$, 51, 137
$\subseteq$, 51, 137
$\mathcal{D}$, 68, 70
$\mathcal{D}_1[\mathcal{D}_2/x]$, 105
$\mathcal{NG}_c^{\Rightarrow,\wedge,\vee,\perp}$, 84
$\mathcal{NG}_i^{\Rightarrow,\wedge,\vee,\perp}$, 85, 102
$\mathcal{NG}_m^{\Rightarrow,\wedge,\vee,\perp}$, 85
$\mathcal{NG}_c^{\Rightarrow,\vee,\wedge,\perp,\forall,\exists}$, 121
$\mathcal{NG}_i^{\Rightarrow,\vee,\wedge,\perp,\forall,\exists}$, 121
$\mathcal{NG}_m^{\Rightarrow}$, 79
$\mathcal{N}_c^{\Rightarrow,\wedge,\vee,\perp}$, 84
$\mathcal{N}_i^{\Rightarrow,\wedge,\vee,\perp}$, 84, 102
$\mathcal{N}_m^{\Rightarrow,\wedge,\vee,\perp}$, 84
$\mathcal{N}_c^{\Rightarrow,\vee,\wedge,\perp,\forall,\exists}$, 120
$\mathcal{N}_i^{\Rightarrow,\vee,\wedge,\perp,\forall,\exists}$, 120
$\mathcal{N}_m^{\Rightarrow}$, 70
$\mathcal{P}(A)$, 54, 137
$\tau_1 = \tau_2$, 117, 118
$\top$, 13, 65, 118, 153
$\vdash P$, 85
$\vdash \Gamma \triangleright M : \sigma$, 109
$\vdash \Gamma \to P$, 85
$\{a, b\}$, 136
$\{a\}$, 137
$\{x \mid x \notin x\}$, 135
$\{x \in A \mid P\}$, 138
$a \in A$, 50, 136
$a \notin A$, 50, 136
a^b, 39
$f(\tau_1, \ldots, \tau_n)$, 117
n^+, 142

Index

α-conversion, 111, 251
  immediate, 251
α-reduction, 251
  immediate, 251
β-conversion, 112, 252
β-normal form, 112
β-redex, 252
β-reduction, 111, 252
  immediate, 111, 252
$\exists$-elimination, 120
$\exists$-introduction, 119
$\forall$-elimination, 119
$\forall$-introduction, 119
$\Rightarrow$-elimination rule, 68, 70, 81
$\Rightarrow$-introduction rule, 68, 70, 81
λ-abstraction, 107, 247
λ-calculus, 106, 247
  pure, 247
  simply-typed, 106, 108
  untyped, 247
λ-term, 107
  β-irreducible, 112
  closed, 107, 249
  raw, 107
  raw simply-typed, 107, 247
  simply-typed, 107
$\wedge$-elimination rule, 82
$\wedge$-introduction rule, 82
$\vee$-elimination rule, 82
$\vee$-introduction rule, 82
$\neg$-elimination rule, 84
$\neg$-introduction rule, 83, 90
$\neg\neg$-elimination rule, 91
$\perp$-elimination rule, 83, 91

absurdity, 13, 65
absurdum, 13, 65
alphabet, 44, 116
And–Elim, 26
And–Intro, 26
Andrews, P., 63, 113
application, 107, 247
associativity, 35, 53, 60, 96, 139, 150
assumption, 18, 66
atomic propositions, 16, 69
auxiliary lemmas, 19, 36, 71, 96, 105
axiom of choice, 143
axiom of infinity, 141, 142, 148
axioms, 11, 18, 67, 69, 81, 158
  of separation, 138
  of set theory, 136

Bernays, P., 134
bound variable, 43, 116
Brouwer, L., 87

Cantor, G., 134
capture of variables, 45, 119
cardinal
  number, 141
Church's theorem, 103, 147
Church, A., 103, 106
classical logic, 85
classical propositional logic, 34
closed, 68
combinator, 249
complement, 139

- of a set, 52
completeness, 41, 42, 58, 99, 100, 102, 114, 146
complexity, 11
- theory, 113
composite, 16
compound statement, 13, 65
comprehension axioms, 138
computation, 63, 106
conclusion, 18
confluence, 112
congruent, 397
conjunction, 12, 13, 64, 65
consistency, 41, 58, 99, 114, 134, 146
consistent contexts, 79
constant symbols, 44, 116
constructivists, 88
constructivity, 87
context, 79, 108
counterexample, 42, 58, 100, 103, 114, 146
Curry, H., 106
Curry–Howard isomorphism, 106, 147
cut-elimination theorem, 105

de Morgan laws, 30, 34, 58, 59, 93, 139, 146, 148
- for quantifiers, 49, 123
decision problem, 103
deduction, 11, 18, 57, 66, 145, 158
- tree, 70, 79, 84, 85, 121
definition, 15
derivation, 11
discharged, 68, 71, 79, 84, 145
disjoint sets, 52
disjunction, 12, 13, 64, 65
distributivity, 35, 53, 60, 96, 139, 150
divisor, 15
double-negation
- elimination, 91, 146
- rule, 88, 145
- translation, 127
dynamic logic, 115

eigenvariable, 120, 121
- restriction, 147
empty, 51, 136
- set, 51, 136, 147
- set axiom, 136
Enderton, H., 134, 143
equality predicate, 117
equation, 117
equivalence
- logical, 14, 33, 66, 85
Equivalence–Intro, 33
even, 15, 130, 148
exclusive or, 154
Exist–Elim, 46
Exist–Intro, 46
extensionality, 50
extensionality axiom, 50, 136

falsity, 13, 65
falsum, 13, 65
finite
- tree, 103
first-order, 117
- classical logic, 120
- formulae, 118
- intuitionistic logic, 120
- language, 117
- logic, 58, 146
- structures, 114
- theories, 147
- theory, 128
- variables, 117
Forall–Elim, 45
Forall–Intro, 45
formula, 44
formulae-as-types principle, 106
Fraenkel, A., 134
free variable, 43, 116
function
- operator, 44, 116
- symbols, 44, 116

Gödel's completeness theorem, 114, 147

Gödel's incompleteness theorem, 129, 147
Gödel, K., 127
Gallier, J., 42, 64, 100, 113, 114, 118, 128
GB, 134
gcd, 39
Gentzen
  sequent, 78, 84, 157
  sequent-calculus, 157
  system, 145
Gentzen, G., 67, 78, 105, 127, 157
Girard, J.Y., 115, 128
Glivenko, V., 127
Gowers, T., 35, 64, 96

Halmos, P., 134
Hilbert, D., 67, 136
Howard, W., 106
hypotheses, 11, 18, 66

if and only if, 14, 66
iff, 14, 66
implication, 12, 13, 57, 64, 65, 145
Implication–Elim, 18
Implication–Intro, 18
implicational logic, 69
inconsistent, 90
induction, 37, 95
  axiom, 129
induction principle, 56, 143
  for $\mathbb{N}$, 56, 59, 143, 148
inductive, 55, 142
  set, 55, 142, 148
inference rules, 12, 57, 64, 67, 145, 158
  for the quantifiers, 119
  for the quantifiers in Gentzen-sequent style, 120
infinite
  intersections, 55, 140
integer, 36
intersection, 51, 138, 147
  of sets, 51, 138
intuitionistic logic, 84, 92
intuitionistically
  provable, 85, 121
  valid, 102
intuitionists, 88
irrational, 37, 38
  number, 37

judgement, 108

Kleene, S., 113
Kolmogorov, A., 127
Kripke, 92
  models, 92, 100, 101, 114, 146
  semantics, 92, 100
Kripke, S., 100

law of the excluded middle, 88, 145
linear
  logic, 115
logic
  classical, 84, 92, 120, 121
  intuitionistic, 84, 85, 120, 121
  mathematical, 63
logical
  connectives, 12, 64
  equivalence, 33, 85
  formula, 13, 65
  language, 44, 117

mathematical, 11
minimal
  logic, 70, 84
modular arithmetic, 397, 398
modus ponens, 18, 36, 68, 70, 81
multiple, 15
multiset, 68, 79

natural
  deduction, 34, 145
  deduction system, 63, 67, 145
  numbers, 36, 142, 148, 164
negation, 12, 13, 64, 65
Negation–Elim, 23
Negation–Intro, 22
nonconstructive proofs, 39, 49, 92, 122

nonlogical symbols, 117
normal form, 92, 105, 106
normalization step, 105
NP-complete, 99
NP-completeness, 146
number
 even, 15, 17
 odd, 15, 17

odd, 15, 130, 148
Or–Elim, 28
Or–Intro, 28

PA, 128
pairing axiom, 136
Peano arithmetic, 117, 128, 147, 156
Peano, G., 128
Peirce's law, 95, 99, 153
Perp–Elim, 23
power set, 54, 137, 147
power set axiom, 137
Prawitz, D., 67, 105
predicate, 15
 symbols, 44, 116
 terms, 44, 117
premise, 18
premises, 11, 18, 57, 66, 68, 145
 closed, 68
 discharged, 68, 70
prenex-form, 156
prime, 16
principle, 11
 of proof by contradiction, 63
 of the excluded middle, 88
 reasoning, 11
proof, 11, 18, 144, 145, 158
 by-contrapositive rule, 58, 95, 146
 checkers, 12
 classical, 63, 145
 constructive, 63, 88, 92, 145
 direct, 16
 formal, 12, 35, 96
 indirect, 16
 informal, 12, 35, 96
 intuitionistic, 145
 mathematical, 11
 minimal, 80
 nonconstructive, 39, 49, 58, 92, 122, 145, 146
 normalization, 77, 100, 105, 106, 147
 principles, 64
 rules, 12, 64, 67
 substitution, 77
 system, 12, 64, 67
 template, 16
 templates, 11, 57
 tree, 70, 79, 84, 85, 121
proof–by–cases, 28, 86
proof–by–contradiction, 58, 145
 for negated propositions, 26
 rule, 23, 83
Proof–By–Contradiction Principle, 23
proof–by–contrapositive, 33
 principle, 33
proper
 subset, 51, 137
propositional
 intuitionistic logic, 85
 logic, 57, 145
 minimal logic, 85
 symbols, 16, 69, 117
propositions, 13, 16, 65, 69
 atomic, 16, 69
provability, 66
provable, 85
 in classical logic, 121

quantified formulae, 117, 118
quantifier
 existential, 43
 universal, 43
quantifiers, 12, 42, 64, 115

RAA, 23, 58, 83, 90–92, 145
rank, 117
rational, 37

    number, 37
real, 37
    number, 37
redex, 252
reductio ad absurdum rule, 23, 83
reduction step, 105
regularity axioms, 143
relative complement, 52, 138, 147
relatively prime, 39
repeated squaring, 412
replacement axioms, 143
rules, 11, 70
    logical, 11, 144
    of logic, 11
Russell's paradox, 135, 147
Russell, B., 135, 140

SAT, 98
satisfiability, 146
    problem, 98, 146
satisfiable, 98
semantics, 39, 97
    truth-value, 39, 58, 97, 146
sequent, 78, 79
    Gentzen, 79
sequents, 145, 157
set, 50, 136
    of integers, 14
    of natural numbers, 14
    of worlds, 101
    theory, 134
set theory, 136
    first-order theory, 136
singleton set, 137
soundness, 41, 58, 99, 102, 114, 146
standard model, 134
statements, 12, 64
    atomic, 12, 64
    compound, 12, 64
Statman, R., 113
strong
    normalization, 106, 147
strongly normalizing, 112
subset, 51, 137
    axioms, 138, 147
substitution, 44, 110, 118, 250
    safe, 111
Suppes, P., 134

Takeuti, G., 103
tautology, 41, 58, 98, 146
temporal logic, 115
terms, 44, 116, 117
theorem provers, 12, 103
theory, 128
    of computation, 103
    of equality, 128
transitive
    set, 163
tree, 68
truth, 13, 39, 65, 66, 97
    tables, 40, 41, 58, 97, 98, 146
    value, 39, 97
        of a proposition, 40, 98
truth-value semantics, 58, 146
Turing machines, 103
Turing, A., 103
type, 106, 273
    second order, 273
    atomic, 106
    base, 106
    polymorphic, 273
    simple, 106
type-assignment, 108
type-checking, 108
    rules, 108

undecidability, 103
    of the decision problem, 103, 147
    of the halting problem, 103
    of the Post correspondence problem, 103
union, 51, 137, 147
    of sets, 51, 137
union axiom, 137, 139
unsatisfiability, 146

unsatisfiable, 98

valid, 41, 98
validity, 58, 146
  problem, 98, 146
van Dalen, D., 42, 64, 92, 100, 102, 114, 115, 118
variable, 107, 247
  bound, 107, 249
  capture, 111, 251
  free, 107, 249
variables
  bound, 58, 146
  free, 58, 146
VNB, 134
von Neumann, J., 134, 142, 164

weakening rule, 74
well-ordering
  of $\mathbb{N}$, 57
witness, 121
  property, 121

Zermelo, E., 134
Zermelo–Fraenkel set theory, 134, 144, 147
ZF, 134, 144, 147