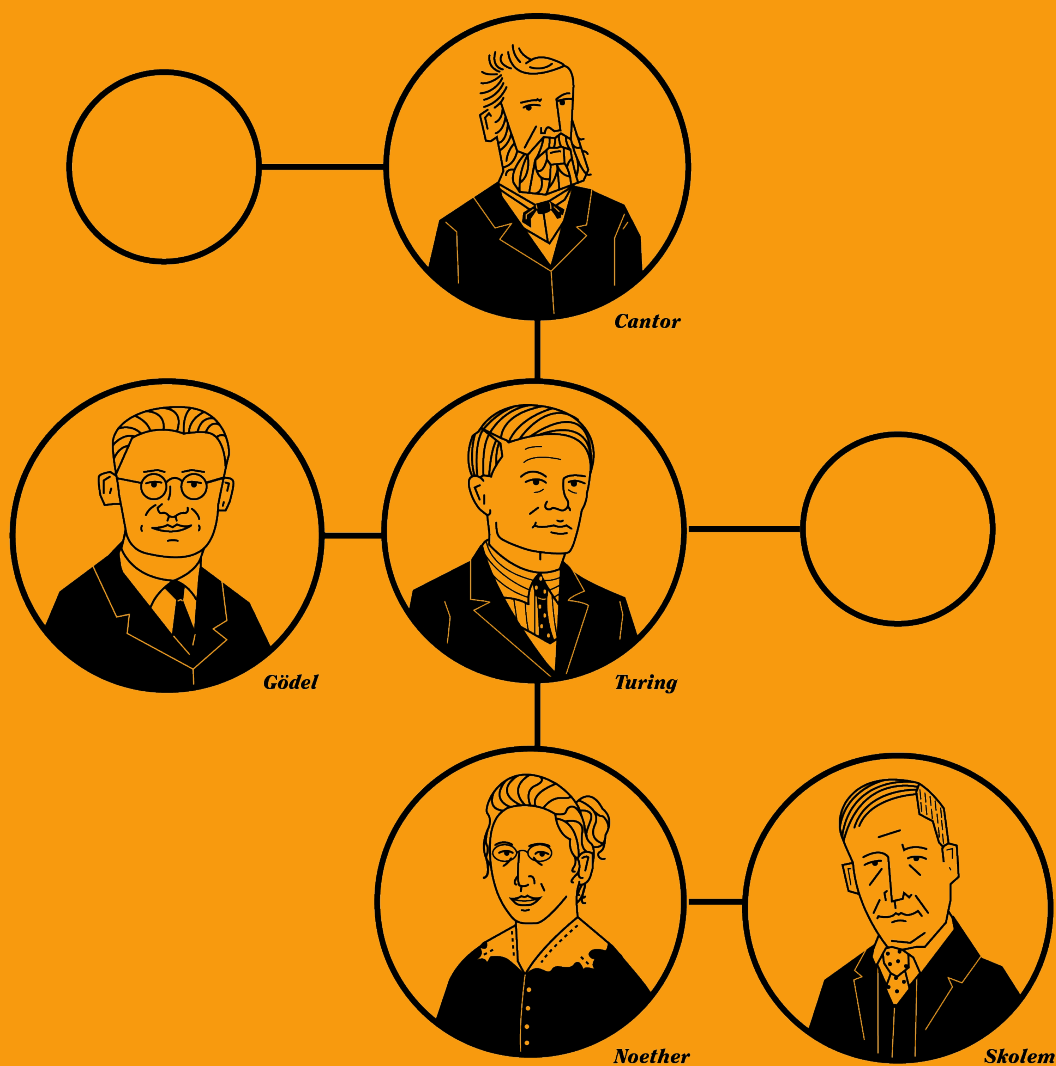


Sets, Logic, Computation

An Open Logic Text



Sets, Logic, Computation

The Open Logic Project

Instigator

Richard Zach, *University of Calgary*

Editorial Board

Aldo Antonelli,[†] *University of California, Davis*

Andrew Arana, *Université Paris I Panthéon–Sorbonne*

Jeremy Avigad, *Carnegie Mellon University*

Walter Dean, *University of Warwick*

Gillian Russell, *University of North Carolina*

Nicole Wyatt, *University of Calgary*

Audrey Yap, *University of Victoria*

Sets, Logic, Computation

An Open Logic Text

Edited by Richard Zach

WINTER 2016

The Open Logic Project would like to acknowledge the generous support of the Faculty of Arts of the University of Calgary and the Alberta OER Initiative.



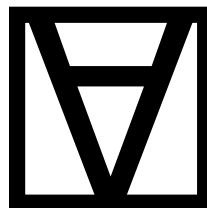
UNIVERSITY OF CALGARY
FACULTY OF ARTS

Illustrations by Matthew Leadbeater, used under a Creative Commons Attribution-NonCommercial 4.0 International License.

Typeset in Baskervald X and Universalis ADF Standard by L^AT_EX.



Sets, Logic, Computation by Richard Zach is licensed under a Creative Commons Attribution 4.0 International License. It is based on *The Open Logic Text* by the Open Logic Project, used under a Creative Commons Attribution 4.0 International License.



Contents

Preface	xi
I Sets, Relations, Functions	1
1 Sets	2
1.1 Basics	2
1.2 Some Important Sets	4
1.3 Subsets	5
1.4 Unions and Intersections	6
1.5 Proofs about Sets	7
1.6 Pairs, Tuples, Cartesian Products	9
2 Relations	11
2.1 Relations as Sets	11
2.2 Special Properties of Relations	13
2.3 Orders	14
2.4 Graphs	16
2.5 Operations on Relations	17
3 Functions	19
3.1 Basics	19
3.2 Kinds of Functions	21
3.3 Inverses of Functions	22

3.4	Composition of Functions	23
3.5	Isomorphism	23
3.6	Partial Functions	24
3.7	Functions and Relations	24
4	The Size of Sets	26
4.1	Introduction	26
4.2	Countable Sets	26
4.3	Uncountable Sets	30
4.4	Reduction	32
4.5	Equinumerous Sets	33
4.6	Comparing Sizes of Sets	35
II	First-order Logic	39
5	Syntax and Semantics	40
5.1	First-Order Languages	40
5.2	Terms and Formulas	43
5.3	Unique Readability	45
5.4	Main operator of a Formula	49
5.5	Subformulas	50
5.6	Free Variables and Sentences	52
5.7	Substitution	53
5.8	Structures for First-order Languages	54
5.9	Satisfaction of a Formula in a Structure	58
5.10	Extensionality	62
5.11	Semantic Notions	64
6	Theories and Their Models	66
6.1	Introduction	66
6.2	Expressing Properties of Structures	68
6.3	Examples of First-Order Theories	70
6.4	Expressing Relations in a Structure	73
6.5	The Theory of Sets	74
6.6	Expressing the Size of Structures	77

7	Natural Deduction	79
7.1	Rules and Derivations	79
7.2	Examples of Derivations	82
7.3	Proof-Theoretic Notions	87
7.4	Properties of Derivability	88
7.5	Soundness	91
7.6	Derivations with Identity predicate	95
8	The Completeness Theorem	97
8.1	Introduction	97
8.2	Outline of the Proof	98
8.3	Maximally Consistent Sets of Sentences	100
8.4	Henkin Expansion	102
8.5	Lindenbaum's Lemma	104
8.6	Construction of a Model	105
8.7	Identity	107
8.8	The Completeness Theorem	110
8.9	The Compactness Theorem	111
8.10	The Löwenheim-Skolem Theorem	111
9	Beyond First-order Logic	113
9.1	Overview	113
9.2	Many-Sorted Logic	114
9.3	Second-Order logic	116
9.4	Higher-Order logic	121
9.5	Intuitionistic logic	124
9.6	Modal Logics	130
9.7	Other Logics	132
III	Turing Machines	135
10	Turing Machine Computations	136
10.1	Introduction	136
10.2	Representing Turing Machines	139
10.3	Turing Machines	144

10.4	Configurations and Computations	145
10.5	Unary Representation of Numbers	147
10.6	Halting States	148
10.7	Combining Turing Machines	149
10.8	Variants of Turing Machines	151
10.9	The Church-Turing Thesis	152
11	Undecidability	154
11.1	Introduction	154
11.2	Enumerating Turing Machines	156
11.3	The Halting Problem	158
11.4	The Decision Problem	160
11.5	Representing Turing Machines	161
11.6	Verifying the Representation	164
11.7	The Decision Problem is Unsolvable	168
A	Induction	171
A.1	Induction on \mathbb{N}	171
A.2	Strong Induction	175
A.3	Inductive Definitions	176
A.4	Structural Induction	178
B	Biographies	180
B.1	Georg Cantor	180
B.2	Alonzo Church	181
B.3	Gerhard Gentzen	182
B.4	Kurt Gödel	184
B.5	Emmy Noether	186
B.6	Bertrand Russell	187
B.7	Alfred Tarski	189
B.8	Alan Turing	190
B.9	Ernst Zermelo	192
C	Problems	194
	Photo Credits	205

Bibliography**207**



Thoralf Skolem

1887 - 1963

Preface

This book is an introduction to meta-logic, aimed especially at students of computer science and philosophy. “Meta-logic” is so-called because it is the discipline that studies logic itself. Logic proper is concerned with canons of valid inference, and its symbolic or formal version presents these canons using formal languages, such as those of propositional and predicate, a.k.a., first-order logic. Meta-logic investigates the properties of these language, and of the canons of correct inference that use them. It studies topics such as how to give precise meaning to the expressions of these formal languages, how to justify the canons of valid inference, what the properties of various proof systems are, including their computational properties. These questions are important and interesting in their own right, because the languages and proof systems investigated are applied in many different areas—in mathematics, philosophy, computer science, and linguistics, especially—but they also serve as examples of how to study formal systems in general. The logical languages we study here are not the only ones people are interested in. For instance, linguists and philosophers are interested in languages that are much more complicated than those of propositional and first-order logic, and computer scientists are interested in other *kinds* of languages altogether, such as programming languages. And the methods we discuss here—how to give semantics for formal languages, how to prove results about formal languages, how

to investigate the properties of formal languages—are applicable in those cases as well.

Like any discipline, meta-logic both has a set of results or facts, and a store of methods and techniques, and this text covers both. Some students won't need to know some of the of the results we discuss outside of this course, but they will need and use the methods we use to establish them. The Löwenheim-Skolem theorem, say, does not often make an appearance in computer science, but the methods we use to prove it, do. On the other hand, many of the results we discuss do have relevance for certain debates, say, in the philosophy of science and in metaphysics. Philosophy students may not need to be able to prove these results after this course, but they do need to know what the results are—and you really only *understand* these results if you have thought through the definitions and proofs needed to finally prove them. These are, in part, the reasons for why the material in this text—both the results and the methods—are recommended, and in some cases required for students of computer science and philosophy.

Subsequent chapters are divided into three parts. Part 1 concerns itself with the theory of sets. Logic and meta-logic is historically connected very closely to what's called the “foundations of mathematics.” Mathematical foundations deal with how ultimately mathematical objects such as integers, rational, and real numbers, functions, spaces, etc., should be understood. Set theory provides one answer (there are others), and so set theory and logic have long been closely connected. Sets, relations, and functions are also ubiquitous in any sort of formal investigation, not just in mathematics but as well in computer science and in some of the more technical corners of philosophy. Certainly for the purposes of formulating and proving results about the semantics and proof theory of logic and the foundation of computability it is essential to have a language in which to do this. For instance, we will talk about sets of expressions, relations of consequence and provability, interpretations of predicate symbols (which turn out to be relations), computable functions, and various relations between and constructions using these. It will just be good to

have shorthand symbols for these, and think through the general properties of sets, relations, and functions in order to do that. If you are not used to thinking mathematically and to formulating mathematical proofs, then think of the first part on set theory as a training ground: all the basic definitions will be given, and we'll give increasingly complicated proofs using them. Note that understanding these proofs—and being able to find and formulate them yourself—is perhaps more important than understanding the results, and especially in the first part, and especially if you are new to mathematical thinking, it is important that you think through the examples and do the problems in [appendix C](#).

In the first part we will establish one important result, however. This result—Cantor's theorem—relies on one of the most striking examples of conceptual analysis to be found anywhere in the sciences, namely, Cantor's analysis of infinity. Infinity has puzzled mathematicians and philosophers alike for centuries; no-one knew how to properly think about it, and many people thought it was a mistake to think about it at all, that the notion of an infinite object or infinite collection itself was incoherent. Cantor made infinity into a subject we can coherently work with, and developed an entire theory of infinite collections—and infinite numbers with which we can measure the sizes of infinite collections—and showed that there are different levels of infinity. This theory of “transfinite” numbers is beautiful and intricate, and we won't get very far into it; but we will be able to show that there are different levels of infinity, specifically, that there are “countable” and “uncountable” levels of infinity. This result does have important applications, but it is also really the kind of result that any self-respecting mathematician, computer scientist, or philosopher should know.

In the second part we turn to first-order logic. We will define the language of first-order logic and its semantics, i.e., what first-order structures are and when a sentence of first-order logic is true in a structure. This will enable us to do two important things: (1) We can define, with mathematical precision, when a sentence is a logical consequence of another. (2) We can also consider how

the relations that make up a first-order structure are described—characterized—by the sentences that are true in them. This in particular leads us to a discussion of the axiomatic method, in which sentences of first-order languages are used to characterize certain kinds of structures. Proof theory will occupy us next, and we will consider the original version of natural deduction as defined in the 1930s by Gerhard Gentzen. The semantic notion of consequence and the syntactic notion of provability give us two completely different ways to make precise the idea that a sentence may follow from some others. The soundness and completeness theorems link these two characterizations. In particular, we will prove Gödel's completeness theorem, which states that whenever a sentence is a semantic consequence of some others, there is also a deduction of said sentence from these others. An equivalent formulation is: if a collection of sentences is consistent—in the sense that nothing contradictory can be proved from them—then there is a structure that makes all of them true.

The second formulation of the completeness theorem is perhaps the more surprising. Around the time Gödel proved this result (in 1929), the German mathematician David Hilbert famously held the view that consistency (i.e., freedom from contradiction) is all that mathematical existence requires. In other words, whenever a mathematician can coherently describe a structure or class of structures, then they should be entitled to believe in the existence of such structures. At the time, many found this idea preposterous: just because you can describe a structure without contradicting yourself, it surely does not follow that such a structure actually exists. But that is exactly what Gödel's completeness theorem says. In addition to this paradoxical—and certainly philosophically intriguing aspect—the completeness theorem also has two important applications which allow us to prove further results about the existence of structures which make given sentences true. These are the compactness and the Löwenheim-Skolem theorem.

In the third part, we connect logic with computability. Again, there is a historical connection: David Hilbert had posed as a

fundamental problem of logic to find a mechanical method which would decide, of a given sentence of logic, whether it has a proof. Such a method exists, of course, for propositional logic: one just has to check all truth tables, and since there are only finitely many of them, the method eventually yields a correct answer. Such a straightforward method is not possible for first-order logic, since the number of possible structures is infinite (and structures themselves may be infinite). Logicians were working on finding a more ingenious method for years. Two logicians—Alonzo Church and Alan Turing—eventually established that there is no such method. In order to do this, it was necessary again to first provide a precise definition of what a mechanical method is in general. If an effective method had been proposed, anyone would have recognized it as an effective method. To prove that no effective method exists, you have to define “effective method” first and give an impossibility proof on the basis of that definition. This is what Turing did: he proposed the idea of a Turing machine¹ as a mathematical model of what a mechanical procedure can, in principle, do. This is another example of a *conceptual analysis* of an informal concept using mathematical machinery; and it is perhaps of the same order of importance for computer science as Cantor’s analysis of infinity is for mathematics. Our last major undertaking will be the proof of two impossibility theorems: we will show that the so-called “halting problem” cannot be solved by Turing machines, and finally that Hilbert’s “decision problem” (for logic) also cannot.

This text is mathematical, in the sense that we discuss mathematical definitions and prove our results mathematically. But it is not mathematical in the sense that you need extensive mathematical background knowledge. Nothing in this text requires advanced knowledge of algebra, trigonometry, or calculus. We have made a special effort to also not require any familiarity with the way mathematics works: in fact, part of the point is to *develop* the kinds of reasoning and proof skills required to understand

¹Turing of course did not call it that himself.

and prove our results. The organization of the text follows mathematical convention, for one reason: these conventions have been developed because clarity and precision are especially important, and so, e.g., it is critical to know when something is asserted as the conclusion of an argument, is offered as a reason for something else, or is intended to introduce new vocabulary. So we follow mathematical convention and label passages as “definitions” if they are used to introduce new terminology or symbols or as “theorems,” “propositions,” “lemmas,” or “corollaries” when we record a result or finding.² Other than these conventions, we will only use the methods of logical proof as they should be familiar from a first logic course, with one exception: we will make extensive use of the method of *induction* to prove results. [Appendix A](#) is devoted to this principle.

²The difference between the latter four is not terribly important, but roughly: A theorem is an important result. A proposition is a result worth recording, but perhaps not as important as a theorem. A lemma is a result we mainly record only because we want to break up a proof into smaller, easier to manage chunks. A corollary is a result that follows easily from a theorem or proposition, such as an interesting special case.



Georg Cantor

1845 - 1918

PART I

Sets, Relations, Functions

CHAPTER 1

Sets

1.1 Basics

Sets are the most fundamental building blocks of mathematical objects. In fact, almost every mathematical object can be seen as a set of some kind. In logic, as in other parts of mathematics, sets and set theoretical talk is ubiquitous. So it will be important to discuss what sets are, and introduce the notations necessary to talk about sets and operations on sets in a standard way.

Definition 1.1. A *set* is a collection of objects, considered independently of the way it is specified, of the order of the objects in the set, or of their multiplicity. The objects making up the set are called *elements* or *members* of the set. If a is an element of a set X , we write $a \in X$ (otherwise, $a \notin X$). The set which has no elements is called the *empty* set and denoted by the symbol \emptyset .

Example 1.2. Whenever you have a bunch of objects, you can collect them together in a set. The set of Richard's siblings, for instance, is a set that contains one person, and we could write it as $S = \{\text{Ruth}\}$. In general, when we have some objects a_1, \dots, a_n , then the set consisting of exactly those objects is written $\{a_1, \dots, a_n\}$. Frequently we'll specify a set by some property that its elements share—as we just did, for instance, by specifying S as the set of Richard's siblings. We'll use the following shorthand

notation for that: $\{x : \dots x \dots\}$, where the $\dots x \dots$ stands for the property that x has to have in order to be counted among the elements of the set. In our example, we could have specified S also as

$$S = \{x : x \text{ is a sibling of Richard}\}.$$

When we say that sets are independent of the way they are specified, we mean that the elements of a set are all that matters. For instance, it so happens that

$$\begin{aligned} &\{\text{Nicole, Jacob}\}, \\ &\{x : x \text{ is a niece or nephew of Richard}\}, \text{ and} \\ &\{x : x \text{ is a child of Ruth}\} \end{aligned}$$

are three ways of specifying one and the same set.

Saying that sets are considered independently of the order of their elements and their multiplicity is a fancy way of saying that

$$\begin{aligned} &\{\text{Nicole, Jacob}\} \text{ and} \\ &\{\text{Jacob, Nicole}\} \end{aligned}$$

are two ways of specifying the same set; and that

$$\begin{aligned} &\{\text{Nicole, Jacob}\} \text{ and} \\ &\{\text{Jacob, Nicole, Nicole}\} \end{aligned}$$

are also two ways of specifying the same set. In other words, all that matters is which elements a set has. The elements of a set are not ordered and each element occurs only once. When we *specify* or *describe* a set, elements may occur multiple times and in different orders, but any descriptions that only differ in the order of elements or in how many times elements are listed describes the same set.

Definition 1.3 (Extensionality). If X and Y are sets, then X and Y are *identical*, $X = Y$, iff every element of X is also an element of Y , and vice versa.

Extensionality gives us a way for showing that sets are identical: to show that $X = Y$, show that whenever $x \in X$ then also $x \in Y$, and whenever $y \in Y$ then also $y \in X$.

1.2 Some Important Sets

Example 1.4. Mostly we'll be dealing with sets that have mathematical objects as members. You will remember the various sets of numbers: \mathbb{N} is the set of *natural* numbers $\{0, 1, 2, 3, \dots\}$; \mathbb{Z} the set of *integers*,

$$\{\dots, -3, -2, -1, 0, 1, 2, 3, \dots\};$$

\mathbb{Q} the set of *rational*s ($\mathbb{Q} = \{z/n : z \in \mathbb{Z}, n \in \mathbb{N}, n \neq 0\}$); and \mathbb{R} the set of *real* numbers. These are all *infinite* sets, that is, they each have infinitely many elements. As it turns out, \mathbb{N} , \mathbb{Z} , \mathbb{Q} have the same number of elements, while \mathbb{R} has a whole bunch more— \mathbb{N} , \mathbb{Z} , \mathbb{Q} are “countable and infinite” whereas \mathbb{R} is “uncountable”.

We'll sometimes also use the set of positive integers $\mathbb{Z}^+ = \{1, 2, 3, \dots\}$ and the set containing just the first two natural numbers $\mathbb{B} = \{0, 1\}$.

Example 1.5 (Strings). Another interesting example is the set A^* of *finite strings* over an alphabet A : any finite sequence of elements of A is a string over A . We include the *empty string* Λ among the strings over A , for every alphabet A . For instance,

$$\begin{aligned} \mathbb{B}^* = \{ \Lambda, 0, 1, 00, 01, 10, 11, \\ 000, 001, 010, 011, 100, 101, 110, 111, 0000, \dots \}. \end{aligned}$$

If $x = x_1 \dots x_n \in A^*$ is a string consisting of n “letters” from A , then we say *length* of the string is n and write $\text{len}(x) = n$.

Example 1.6 (Infinite sequences). For any set A we may also consider the set A^ω of infinite sequences of elements of A . An infinite sequence $a_1 a_2 a_3 a_4 \dots$ consists of a one-way infinite list of objects, each one of which is an element of A .

1.3 Subsets

Sets are made up of their elements, and every element of a set is a part of that set. But there is also a sense that some of the elements of a set *taken together* are a “part of” that set. For instance, the number 2 is part of the set of integers, but the set of even numbers is also a part of the set of integers. It’s important to keep those two senses of being part of a set separate.

Definition 1.7. If every element of a set X is also an element of Y , then we say that X is a *subset* of Y , and write $X \subseteq Y$.

Example 1.8. First of all, every set is a subset of itself, and \emptyset is a subset of every set. The set of even numbers is a subset of the set of natural numbers. Also, $\{a, b\} \subseteq \{a, b, c\}$.

But $\{a, b, e\}$ is not a subset of $\{a, b, c\}$.

Note that a set may contain other sets, not just as subsets but as elements! In particular, a set may happen to *both* be an element and a subset of another, e.g., $\{0\} \in \{0, \{0\}\}$ and also $\{0\} \subseteq \{0, \{0\}\}$.

Extensionality gives a criterion of identity for sets: $X = Y$ iff every element of X is also an element of Y and vice versa. The definition of “subset” defines $X \subseteq Y$ precisely as the first half of this criterion: every element of X is also an element of Y . Of course the definition also applies if we switch X and Y : $Y \subseteq X$ iff every element of Y is also an element of X . And that, in turn, is exactly the “vice versa” part of extensionality. In other words, extensionality amounts to: $X = Y$ iff $X \subseteq Y$ and $Y \subseteq X$.

Definition 1.9. The set consisting of all subsets of a set X is called the *power set of* X , written $\wp(X)$.

$$\wp(X) = \{x : x \subseteq X\}$$

Example 1.10. What are all the possible subsets of $\{a, b, c\}$? They are: \emptyset , $\{a\}$, $\{b\}$, $\{c\}$, $\{a, b\}$, $\{a, c\}$, $\{b, c\}$, $\{a, b, c\}$. The set of all these subsets is $\wp(\{a, b, c\})$:

$$\wp(\{a, b, c\}) = \{\emptyset, \{a\}, \{b\}, \{c\}, \{a, b\}, \{b, c\}, \{a, c\}, \{a, b, c\}\}$$

1.4 Unions and Intersections

Definition 1.11. The *union* of two sets X and Y , written $X \cup Y$, is the set of all things which are elements of X , Y , or both.

$$X \cup Y = \{x : x \in X \vee x \in Y\}$$

Example 1.12. Since the multiplicity of elements doesn't matter, the union of two sets which have an element in common contains that element only once, e.g., $\{a, b, c\} \cup \{a, 0, 1\} = \{a, b, c, 0, 1\}$.

The union of a set and one of its subsets is just the bigger set: $\{a, b, c\} \cup \{a\} = \{a, b, c\}$.

The union of a set with the empty set is identical to the set: $\{a, b, c\} \cup \emptyset = \{a, b, c\}$.

Definition 1.13. The *intersection* of two sets X and Y , written $X \cap Y$, is the set of all things which are elements of both X and Y .

$$X \cap Y = \{x : x \in X \wedge x \in Y\}$$

Two sets are called *disjoint* if their intersection is empty. This means they have no elements in common.

Example 1.14. If two sets have no elements in common, their intersection is empty: $\{a, b, c\} \cap \{0, 1\} = \emptyset$.

If two sets do have elements in common, their intersection is the set of all those: $\{a, b, c\} \cap \{a, b, d\} = \{a, b\}$.

The intersection of a set with one of its subsets is just the smaller set: $\{a, b, c\} \cap \{a, b\} = \{a, b\}$.

The intersection of any set with the empty set is empty: $\{a, b, c\} \cap \emptyset = \emptyset$.

We can also form the union or intersection of more than two sets. An elegant way of dealing with this in general is the following: suppose you collect all the sets you want to form the union (or intersection) of into a single set. Then we can define the union of all our original sets as the set of all objects which belong to at least one element of the set, and the intersection as the set of all objects which belong to every element of the set.

Definition 1.15. If C is a set of sets, then $\bigcup C$ is the set of elements of elements of C :

$$\begin{aligned}\bigcup C &= \{x : x \text{ belongs to an element of } C\}, \text{ i.e.,} \\ \bigcup C &= \{x : \text{there is a } y \in C \text{ so that } x \in y\}\end{aligned}$$

Definition 1.16. If C is a set of sets, then $\bigcap C$ is the set of objects which all elements of C have in common:

$$\begin{aligned}\bigcap C &= \{x : x \text{ belongs to every element of } C\}, \text{ i.e.,} \\ \bigcap C &= \{x : \text{for all } y \in C, x \in y\}\end{aligned}$$

Example 1.17. Suppose $C = \{\{a, b\}, \{a, d, e\}, \{a, d\}\}$. Then $\bigcup C = \{a, b, d, e\}$ and $\bigcap C = \{a\}$.

We could also do the same for a sequence of sets A_1, A_2, \dots

$$\begin{aligned}\bigcup_i A_i &= \{x : x \text{ belongs to one of the } A_i\} \\ \bigcap_i A_i &= \{x : x \text{ belongs to every } A_i\}.\end{aligned}$$

Definition 1.18. The *difference* $X \setminus Y$ is the set of all elements of X which are not also elements of Y , i.e.,

$$X \setminus Y = \{x : x \in X \text{ and } x \notin Y\}.$$

1.5 Proofs about Sets

Sets and the notations we've introduced so far provide us with convenient shorthands for specifying sets and expressing relationships between them. Often it will also be necessary to prove claims about such relationships. If you're not familiar with mathematical proofs, this may be new to you. So we'll walk through a simple example. We'll prove that for any sets X and Y , it's always the case that $X \cap (X \cup Y) = X$. How do you prove an identity between sets like this? Recall that sets are determined

solely by their elements, i.e., sets are identical iff they have the same elements. So in this case we have to prove that (a) every element of $X \cap (X \cup Y)$ is also an element of X and, conversely, that (b) every element of X is also an element of $X \cap (X \cup Y)$. In other words, we show that both (a) $X \cap (X \cup Y) \subseteq X$ and (b) $X \subseteq X \cap (X \cup Y)$.

A proof of a general claim like “every element z of $X \cap (X \cup Y)$ is also an element of X ” is proved by first assuming that an arbitrary $z \in X \cap (X \cup Y)$ is given, and proving from this assumption that $z \in X$. You may know this pattern as “general conditional proof.” In this proof we’ll also have to make use of the definitions involved in the assumption and conclusion, e.g., in this case of “ \cap ” and “ \cup .” So case (a) would be argued as follows:

(a) We first want to show that $X \cap (X \cup Y) \subseteq X$, i.e., by definition of \subseteq , that if $z \in X \cap (X \cup Y)$ then $z \in X$, for any z . So assume that $z \in X \cap (X \cup Y)$. Since z is an element of the intersection of two sets iff it is an element of both sets, we can conclude that $z \in X$ and also $z \in X \cup Y$. In particular, $z \in X$. But this is what we wanted to show.

This completes the first half of the proof. Note that in the last step we used the fact that if a conjunction ($z \in X$ and $z \in X \cup Y$) follows from an assumption, each conjunct follows from that same assumption. You may know this rule as “conjunction elimination,” or \wedge Elim. Now let’s prove (b):

(b) We now prove that $X \subseteq X \cap (X \cup Y)$, i.e., by definition of \subseteq , that if $z \in X$ then also $z \in X \cap (X \cup Y)$, for any z . Assume $z \in X$. To show that $z \in X \cap (X \cup Y)$, we have to show (by definition of “ \cap ”) that (i) $z \in X$ and also (ii) $z \in X \cup Y$. Here (i) is just our assumption, so there is nothing further to prove. For (ii), recall that z is an element of a union of sets iff it is an element of at least one of those sets. Since

$z \in X$, and $X \cup Y$ is the union of X and Y , this is the case here. So $z \in X \cup Y$. We've shown both (i) $z \in X$ and (ii) $z \in X \cup Y$, hence, by definition of " \cap ," $z \in X \cap (X \cup Y)$.

This was somewhat long-winded, but it illustrates how we reason about sets and their relationships. We usually aren't this explicit; in particular, we might not repeat all the definitions. A "textbook" proof of our result would look something like this.

Proposition 1.19 (Absorption). *For all sets X, Y ,*

$$X \cap (X \cup Y) = X$$

Proof. (a) Suppose $z \in X \cap (X \cup Y)$. Then $z \in X$, so $X \cap (X \cup Y) \subseteq X$.

(b) Now suppose $z \in X$. Then also $z \in X \cup Y$, and therefore also $z \in X \cap (X \cup Y)$. Thus, $X \subseteq X \cap (X \cup Y)$. \square

1.6 Pairs, Tuples, Cartesian Products

Sets have no order to their elements. We just think of them as an unordered collection. So if we want to represent order, we use *ordered pairs* $\langle x, y \rangle$, or more generally, *ordered n -tuples* $\langle x_1, \dots, x_n \rangle$.

Definition 1.20. Given sets X and Y , their *Cartesian product* $X \times Y$ is $\{\langle x, y \rangle : x \in X \text{ and } y \in Y\}$.

Example 1.21. If $X = \{0, 1\}$, and $Y = \{1, a, b\}$, then their product is

$$X \times Y = \{\langle 0, 1 \rangle, \langle 0, a \rangle, \langle 0, b \rangle, \langle 1, 1 \rangle, \langle 1, a \rangle, \langle 1, b \rangle\}.$$

Example 1.22. If X is a set, the product of X with itself, $X \times X$, is also written X^2 . It is the set of *all* pairs $\langle x, y \rangle$ with $x, y \in X$. The set of all triples $\langle x, y, z \rangle$ is X^3 , and so on.

Example 1.23. If X is a set, a *word* over X is any sequence of elements of X . A sequence can be thought of as an n -tuple of elements of X . For instance, if $X = \{a, b, c\}$, then the sequence “ bac ” can be thought of as the triple $\langle b, a, c \rangle$. Words, i.e., sequences of symbols, are of crucial importance in computer science, of course. By convention, we count elements of X as sequences of length 1, and \emptyset as the sequence of length 0. The set of *all* words over X then is

$$X^* = \{\emptyset\} \cup X \cup X^2 \cup X^3 \cup \dots$$

CHAPTER 2

Relations

2.1 Relations as Sets

You will no doubt remember some interesting relations between objects of some of the sets we've mentioned. For instance, numbers come with an *order relation* $<$ and from the theory of whole numbers the relation of *divisibility without remainder* (usually written $n \mid m$) may be familiar. There is also the relation *is identical with* that every object bears to itself and to no other thing. But there are many more interesting relations that we'll encounter, and even more possible relations. Before we review them, we'll just point out that we can look at relations as a special sort of set. For this, first recall what a *pair* is: if a and b are two objects, we can combine them into the *ordered pair* $\langle a, b \rangle$. Note that for ordered pairs the order *does* matter, e.g., $\langle a, b \rangle \neq \langle b, a \rangle$, in contrast to unordered pairs, i.e., 2-element sets, where $\{a, b\} = \{b, a\}$.

If X and Y are sets, then the *Cartesian product* $X \times Y$ of X and Y is the set of all pairs $\langle a, b \rangle$ with $a \in X$ and $b \in Y$. In particular, $X^2 = X \times X$ is the set of all pairs from X .

Now consider a relation on a set, e.g., the $<$ -relation on the set \mathbb{N} of natural numbers, and consider the set of all pairs of numbers $\langle n, m \rangle$ where $n < m$, i.e.,

$$R = \{\langle n, m \rangle : n, m \in \mathbb{N} \text{ and } n < m\}.$$

Then there is a close connection between the number n being

less than a number m and the corresponding pair $\langle n, m \rangle$ being a member of R , namely, $n < m$ if and only if $\langle n, m \rangle \in R$. In a sense we can consider the set R to be the $<$ -relation on the set \mathbb{N} . In the same way we can construct a subset of \mathbb{N}^2 for any relation between numbers. Conversely, given any set of pairs of numbers $S \subseteq \mathbb{N}^2$, there is a corresponding relation between numbers, namely, the relationship n bears to m if and only if $\langle n, m \rangle \in S$. This justifies the following definition:

Definition 2.1. A *binary relation* on a set X is a subset of X^2 . If $R \subseteq X^2$ is a binary relation on X and $x, y \in X$, we write Rxy (or xRy) for $\langle x, y \rangle \in R$.

Example 2.2. The set \mathbb{N}^2 of pairs of natural numbers can be listed in a 2-dimensional matrix like this:

$$\begin{array}{ccccc} \langle 0, 0 \rangle & \langle 0, 1 \rangle & \langle 0, 2 \rangle & \langle 0, 3 \rangle & \dots \\ \langle 1, 0 \rangle & \langle 1, 1 \rangle & \langle 1, 2 \rangle & \langle 1, 3 \rangle & \dots \\ \langle 2, 0 \rangle & \langle 2, 1 \rangle & \langle 2, 2 \rangle & \langle 2, 3 \rangle & \dots \\ \langle 3, 0 \rangle & \langle 3, 1 \rangle & \langle 3, 2 \rangle & \langle 3, 3 \rangle & \dots \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{array}$$

The subset consisting of the pairs lying on the diagonal, i.e.,

$$\{\langle 0, 0 \rangle, \langle 1, 1 \rangle, \langle 2, 2 \rangle, \dots\},$$

is the *identity relation* on \mathbb{N} . (Since the identity relation is popular, let's define $\text{Id}_X = \{\langle x, x \rangle : x \in X\}$ for any set X .) The subset of all pairs lying above the diagonal, i.e.,

$$L = \{\langle 0, 1 \rangle, \langle 0, 2 \rangle, \dots, \langle 1, 2 \rangle, \langle 1, 3 \rangle, \dots, \langle 2, 3 \rangle, \langle 2, 4 \rangle, \dots\},$$

is the *less than* relation, i.e., Lnm iff $n < m$. The subset of pairs below the diagonal, i.e.,

$$G = \{\langle 1, 0 \rangle, \langle 2, 0 \rangle, \langle 2, 1 \rangle, \langle 3, 0 \rangle, \langle 3, 1 \rangle, \langle 3, 2 \rangle, \dots\},$$

is the *greater than* relation, i.e., Gnm iff $n > m$. The union of L with I , $K = L \cup I$, is the *less than or equal to* relation: Knm iff

$n \leq m$. Similarly, $H = G \cup I$ is the *greater than or equal to relation*. L , G , K , and H are special kinds of relations called *orders*. L and G have the property that no number bears L or G to itself (i.e., for all n , neither Lnn nor Gnn). Relations with this property are called *antireflexive*, and, if they also happen to be orders, they are called *strict orders*.

Although orders and identity are important and natural relations, it should be emphasized that according to our definition *any* subset of X^2 is a relation on X , regardless of how unnatural or contrived it seems. In particular, \emptyset is a relation on any set (the *empty relation*, which no pair of elements bears), and X^2 itself is a relation on X as well (one which every pair bears), called the *universal relation*. But also something like $E = \{\langle n, m \rangle : n > 5 \text{ or } m \times n \geq 34\}$ counts as a relation.

2.2 Special Properties of Relations

Some kinds of relations turn out to be so common that they have been given special names. For instance, \leq and \subseteq both relate their respective domains (say, \mathbb{N} in the case of \leq and $\wp(X)$ in the case of \subseteq) in similar ways. To get at exactly how these relations are similar, and how they differ, we categorize them according to some special properties that relations can have. It turns out that (combinations of) some of these special properties are especially important: orders and equivalence relations.

Definition 2.3. A relation $R \subseteq X^2$ is *reflexive* iff, for every $x \in X$, Rxx .

Definition 2.4. A relation $R \subseteq X^2$ is *transitive* iff, whenever Rxy and Ryz , then also Rxz .

Definition 2.5. A relation $R \subseteq X^2$ is *symmetric* iff, whenever Rxy , then also Ryx .

Definition 2.6. A relation $R \subseteq X^2$ is *anti-symmetric* iff, whenever both Rxy and Ryx , then $x = y$ (or, in other words: if $x \neq y$ then either $\neg Rxy$ or $\neg Ryx$).

In a symmetric relation, Rxy and Ryx always hold together, or neither holds. In an anti-symmetric relation, the only way for Rxy and Ryx to hold together is if $x = y$. Note that this does not *require* that Rxy and Ryx holds when $x = y$, only that it isn't ruled out. So an anti-symmetric relation can be reflexive, but it is not the case that every anti-symmetric relation is reflexive. Also note that being anti-symmetric and merely not being symmetric are different conditions. In fact, a relation can be both symmetric and anti-symmetric at the same time (e.g., the identity relation is).

Definition 2.7. A relation $R \subseteq X^2$ is *connected* if for all $x, y \in X$, if $x \neq y$, then either Rxy or Ryx .

Definition 2.8. A relation $R \subseteq X^2$ that is reflexive, transitive, and anti-symmetric is called a *partial order*. A partial order that is also connected is called a *linear order*.

Definition 2.9. A relation $R \subseteq X^2$ that is reflexive, symmetric, and transitive is called an *equivalence relation*.

2.3 Orders

Definition 2.10. A relation which is both reflexive and transitive is called a *preorder*. A preorder which is also anti-symmetric is called a *partial order*. A partial order which is also connected is called a *total order* or *linear order*. (If we want to emphasize that the order is reflexive, we add the adjective “weak”—see below).

Example 2.11. Every linear order is also a partial order, and every partial order is also a preorder, but the converses don't hold. For instance, the identity relation and the full relation on X are preorders, but they are not partial orders, because they are not

anti-symmetric (if X has more than one element). For a somewhat less silly example, consider the *no longer than* relation \preceq on \mathbb{B}^* : $x \preceq y$ iff $\text{len}(x) \leq \text{len}(y)$. This is a preorder, even a linear preorder, but not a partial order.

The relation of *divisibility without remainder* gives us an example of a partial order which isn't a linear order: for integers n, m , we say n (evenly) divides m , in symbols: $n \mid m$, if there is some k so that $m = kn$. On \mathbb{N} , this is a partial order, but not a linear order: for instance, $2 \nmid 3$ and also $3 \nmid 2$. Considered as a relation on \mathbb{Z} , divisibility is only a preorder since anti-symmetry fails: $1 \mid -1$ and $-1 \mid 1$ but $1 \neq -1$. Another important partial order is the relation \subseteq on a set of sets.

Notice that the examples L and G from [Example 2.2](#), although we said there that they were called “strict orders” are not linear orders even though they are connected (they are not reflexive). But there is a close connection, as we will see momentarily.

Definition 2.12. A relation R on X is called *irreflexive* if, for all $x \in X$, $\neg Rxx$. R is called *asymmetric* if for no pair $x, y \in X$ we have Rxy and Ryx . A *strict partial order* is a relation which is irreflexive, asymmetric, and transitive. A strict partial order which is also connected is called a *strict linear order*.

A strict partial order R on X can be turned into a weak partial order R' by adding the identity relation on X : $R' = R \cup \text{Id}_X$. Conversely, starting from a weak partial order, one can get a strict partial order by removing Id_X , i.e., $R' = R \setminus \text{Id}_X$.

Proposition 2.13. R is a strict partial (linear) order on X iff R' is a weak partial (linear) order. Moreover, Rxy iff $R'xy$ for all $x \neq y$.

Example 2.14. \leq is the weak linear order corresponding to the strict linear order $<$. \subseteq is the weak partial order corresponding to the strict partial order \subsetneq .

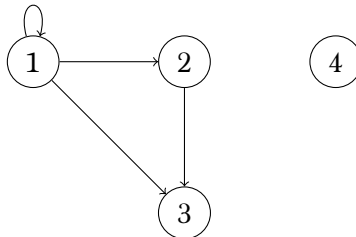
2.4 Graphs

A *graph* is a diagram in which points—called “nodes” or “vertices” (plural of “vertex”)—are connected by edges. Graphs are a ubiquitous tool in discrete mathematics and in computer science. They are incredibly useful for representing, and visualizing, relationships and structures, from concrete things like networks of various kinds to abstract structures such as the possible outcomes of decisions. There are many different kinds of graphs in the literature which differ, e.g., according to whether the edges are directed or not, have labels or not, whether there can be edges from a node to the same node, multiple edges between the same nodes, etc. *Directed graphs* have a special connection to relations.

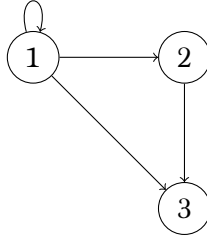
Definition 2.15. A *directed graph* $G = \langle V, E \rangle$ is a set of *vertices* V and a set of *edges* $E \subseteq V^2$.

According to our definition, a graph just is a set together with a relation on that set. Of course, when talking about graphs, it’s only natural to expect that they are graphically represented: we can draw a graph by connecting two vertices v_1 and v_2 by an arrow iff $\langle v_1, v_2 \rangle \in E$. The only difference between a relation by itself and a graph is that a graph specifies the set of vertices, i.e., a graph may have isolated vertices. The important point, however, is that every relation R on a set X can be seen as a directed graph $\langle X, R \rangle$, and conversely, a directed graph $\langle V, E \rangle$ can be seen as a relation $E \subseteq V^2$ with the set V explicitly specified.

Example 2.16. The graph $\langle V, E \rangle$ with $V = \{1, 2, 3, 4\}$ and $E = \{\langle 1, 1 \rangle, \langle 1, 2 \rangle, \langle 1, 3 \rangle, \langle 2, 3 \rangle\}$ looks like this:



This is a different graph than $\langle V', E \rangle$ with $V' = \{1, 2, 3\}$, which looks like this:



2.5 Operations on Relations

It is often useful to modify or combine relations. We've already used the union of relations above (which is just the union of two relations considered as sets of pairs). Here are some other ways:

Definition 2.17. Let $R, S \subseteq X^2$ be relations and Y a set.

1. The *inverse* R^{-1} of R is $R^{-1} = \{\langle y, x \rangle : \langle x, y \rangle \in R\}$.
2. The *relative product* $R \mid S$ of R and S is

$$(R \mid S) = \{\langle x, z \rangle : \text{for some } y, Rxy \text{ and } Syz\}$$

3. The *restriction* $R \upharpoonright Y$ of R to Y is $R \cap Y^2$
4. The *application* $R[Y]$ of R to Y is

$$R[Y] = \{y : \text{for some } x \in X, Rxy\}$$

Example 2.18. Let $S \subseteq \mathbb{Z}^2$ be the successor relation on \mathbb{Z} , i.e., the set of pairs $\langle x, y \rangle$ where $x + 1 = y$, for $x, y \in \mathbb{Z}$. Sxy holds iff y is the successor of x .

1. The inverse S^{-1} of S is the predecessor relation, i.e., $S^{-1}xy$ iff $x - 1 = y$.

2. The relative product $S \mid S$ is the relation x bears to y if $x + 2 = y$.
3. The restriction of S to \mathbb{N} is the successor relation on \mathbb{N} .
4. The application of S to a set, e.g., $S[\{1, 2, 3\}]$ is $\{2, 3, 4\}$.

Definition 2.19. The *transitive closure* R^+ of a relation $R \subseteq X^2$ is $R^+ = \bigcup_{i=1}^{\infty} R^i$ where $R^1 = R$ and $R^{i+1} = R^i \mid R$.

The *reflexive transitive closure* of R is $R^* = R^+ \cup I_X$.

Example 2.20. Take the successor relation $S \subseteq \mathbb{Z}^2$. S^2xy iff $x + 2 = y$, S^3xy iff $x + 3 = y$, etc. So R^*xy iff for some $i \geq 1$, $x + i = y$. In other words, S^+xy iff $x < y$ (and R^*xy iff $x \leq y$).

CHAPTER 3

Functions

3.1 Basics

A *function* is a mapping of which pairs each object of a given set with a unique partner. For instance, the operation of adding 1 defines a function: each number n is paired with a unique number $n + 1$. More generally, functions may take pairs, triples, etc., of inputs and returns some kind of output. Many functions are familiar to us from basic arithmetic. For instance, addition and multiplication are functions. They take in two numbers and return a third. In this mathematical, abstract sense, a function is a *black box*: what matters is only what output is paired with what input, not the method for calculating the output.

Definition 3.1. A *function* $f: X \rightarrow Y$ is a mapping of each element of X to an element of Y . We call X the *domain* of f and Y the *codomain* of f . The *range* $\text{ran}(f)$ of f is the subset of the codomain that is actually output by f for some input.

Example 3.2. Multiplication takes pairs of natural numbers as inputs and maps them to natural numbers as outputs, so goes from $\mathbb{N} \times \mathbb{N}$ (the domain) to \mathbb{N} (the codomain). As it turns out, the range is also \mathbb{N} , since every $n \in \mathbb{N}$ is $n \times 1$.

Multiplication is a function because it pairs each input—each pair of natural numbers—with a single output: $\times: \mathbb{N}^2 \rightarrow \mathbb{N}$. By

contrast, the square root operation applied to the domain \mathbb{N} is not functional, since each positive integer n has two square roots: \sqrt{n} and $-\sqrt{n}$. We can make it functional by only returning the positive square root: $\sqrt{\cdot} : \mathbb{N} \rightarrow \mathbb{R}$. The relation that pairs each student in a class with their final grade is a function—no student can get two different final grades in the same class. The relation that pairs each student in a class with their parents is not a function—generally each student will have at least two parents.

Example 3.3. Let $f : \mathbb{N} \rightarrow \mathbb{N}$ be defined such that $f(x) = x + 1$. This is a definition that specifies f as a function which takes in natural numbers and outputs natural numbers. It tells us that, given a natural number x , f will output its successor $x + 1$. In this case, the codomain \mathbb{N} is not the range of f , since the natural number 0 is not the successor of any natural number. The range of f is the set of all positive integers, \mathbb{Z}^+ .

Example 3.4. Let $g : \mathbb{N} \rightarrow \mathbb{N}$ be defined such that $g(x) = x + 2 - 1$. This tells us that g is a function which takes in natural numbers and outputs natural numbers. Given a natural number n , g will output the predecessor of the successor of the successor of x , i.e., $x + 1$. Despite their different definitions, g and f are the same function.

Functions f and g defined above are the same because for any natural number x , $x + 2 - 1 = x + 1$. f and g pair each natural number with the same output. The definitions for f and g specify the same mapping by means of different equations, and so count as the same function.

Example 3.5. We can also define functions by cases. For instance, we could define $h : \mathbb{N} \rightarrow \mathbb{N}$ by

$$h(x) = \begin{cases} \frac{x}{2} & \text{if } x \text{ is even} \\ \frac{x+1}{2} & \text{if } x \text{ is odd.} \end{cases}$$

Since every natural number is either even or odd, the output of this function will always be a natural number. Just remember that

if you define a function by cases, every possible input must fall into exactly one case.

3.2 Kinds of Functions

Definition 3.6. A function $f: X \rightarrow Y$ is *surjective* iff Y is also the range of f , i.e., for every $y \in Y$ there is at least one $x \in X$ such that $f(x) = y$.

If you want to show that a function is surjective, then you need to show that every object in the codomain is the output of the function given some input or other.

Definition 3.7. A function $f: X \rightarrow Y$ is *injective* iff for each $y \in Y$ there is at most one $x \in X$ such that $f(x) = y$.

Any function pairs each possible input with a unique output. An injective function has a unique input for each possible output. If you want to show that a function f is injective, you need to show that for any element y of the codomain, if $f(x) = y$ and $f(w) = y$, then $x = w$.

A function which is neither injective, nor surjective, is the constant function $f: \mathbb{N} \rightarrow \mathbb{N}$ where $f(x) = 1$.

A function which is both injective and surjective is the identity function $f: \mathbb{N} \rightarrow \mathbb{N}$ where $f(x) = x$.

The successor function $f: \mathbb{N} \rightarrow \mathbb{N}$ where $f(x) = x + 1$ is injective, but not surjective.

The function

$$f(x) = \begin{cases} \frac{x}{2} & \text{if } x \text{ is even} \\ \frac{x+1}{2} & \text{if } x \text{ is odd.} \end{cases}$$

is surjective, but not injective.

Definition 3.8. A function $f: X \rightarrow Y$ is *bijective* iff it is both surjective and injective. We call such a function a *bijection* from X to Y (or between X and Y).

3.3 Inverses of Functions

One obvious question about functions is whether a given mapping can be “reversed.” For instance, the successor function $f(x) = x + 1$ can be reversed in the sense that the function $g(y) = y - 1$ “undos” what f does. But we must be careful: While the definition of g defines a function $\mathbb{Z} \rightarrow \mathbb{Z}$, it does not define a function $\mathbb{N} \rightarrow \mathbb{N}$ ($g(0) \notin \mathbb{N}$). So even in simple cases, it is not quite obvious if functions can be reversed, and that it may depend on the domain and codomain. Let’s give a precise definition.

Definition 3.9. A function $g: Y \rightarrow X$ is an *inverse* of a function $f: X \rightarrow Y$ if $f(g(y)) = y$ and $g(f(x)) = x$ for all $x \in X$ and $y \in Y$.

When do functions have inverses? A good candidate for an inverse of $f: X \rightarrow Y$ is $g: Y \rightarrow X$ “defined by”

$$g(y) = \text{“the” } x \text{ such that } f(x) = y.$$

The scare quotes around “defined by” suggest that this is not a definition. At least, it is not in general. For in order for this definition to specify a function, there has to be one and only one x such that $f(x) = y$ —the output of g has to be uniquely specified. Moreover, it has to be specified for every $y \in Y$. If there are x_1 and $x_2 \in X$ with $x_1 \neq x_2$ but $f(x_1) = f(x_2)$, then $g(y)$ would not be uniquely specified for $y = f(x_1) = f(x_2)$. And if there is no x at all such that $f(x) = y$, then $g(y)$ is not specified at all. In other words, for g to be defined, f has to be injective and surjective.

Proposition 3.10. *If $f: X \rightarrow Y$ is bijective, f has a unique inverse $f^{-1}: Y \rightarrow X$.*

Proof. Exercise. □

3.4 Composition of Functions

We have already seen that the inverse f^{-1} of a bijective function f is itself a function. It is also possible to compose functions f and g to define a new function by first applying f and then g . Of course, this is only possible if the domains and codomains match, i.e., the codomain of f must be a subset of the domain of g .

Definition 3.11. Let $f: X \rightarrow Y$ and $g: Y \rightarrow Z$. The *composition* of f with g is the function $(g \circ f): X \rightarrow Z$, where $(g \circ f)(x) = g(f(x))$.

The function $(g \circ f): X \rightarrow Z$ pairs each member of X with a member of Z . We specify which member of Z a member of X is paired with as follows—given an input $x \in X$, first apply the function f to x , which will output some $y \in Y$. Then apply the function g to y , which will output some $z \in Z$.

Example 3.12. Consider the functions $f(x) = x + 1$, and $g(x) = 2x$. What function do you get when you compose these two? $(g \circ f)(x) = g(f(x))$. So that means for every natural number you give this function, you first add one, and then you multiply the result by two. So their composition is $(g \circ f)(x) = 2(x + 1)$.

3.5 Isomorphism

An *isomorphism* is a bijection that preserves the structure of the sets it relates, where structure is a matter of the relationships that obtain between the elements of the sets. Consider the following two sets $X = \{1, 2, 3\}$ and $Y = \{4, 5, 6\}$. These sets are both structured by the relations successor, less than, and greater than. An isomorphism between the two sets is a bijection that preserves those structures. So a bijective function $f: X \rightarrow Y$ is an isomorphism if, $i < j$ iff $f(i) < f(j)$, $i > j$ iff $f(i) > f(j)$, and j is the successor of i iff $f(j)$ is the successor of $f(i)$.

Definition 3.13. Let U be the pair $\langle X, R \rangle$ and V be the pair $\langle Y, S \rangle$ such that X and Y are sets and R and S are relations on X and Y respectively. A bijection f from X to Y is an *isomorphism* from U to V iff it preserves the relational structure, that is, for any x_1 and x_2 in X , $\langle x_1, x_2 \rangle \in R$ iff $\langle f(x_1), f(x_2) \rangle \in S$.

Example 3.14. Consider the following two sets $X = \{1, 2, 3\}$ and $Y = \{4, 5, 6\}$, and the relations less than and greater than. The function $f: X \rightarrow Y$ where $f(x) = 7 - x$ is an isomorphism between $\langle X, < \rangle$ and $\langle Y, > \rangle$.

3.6 Partial Functions

It is sometimes useful to relax the definition of function so that it is not required that the output of the function is defined for all possible inputs. Such mappings are called *partial functions*.

Definition 3.15. A *partial function* $f: X \rightarrow Y$ is a mapping which assigns to every element of X at most one element of Y . If f assigns an element of Y to $x \in X$, we say $f(x)$ is *defined*, and otherwise *undefined*. If $f(x)$ is defined, we write $f(x) \downarrow$, otherwise $f(x) \uparrow$. The *domain* of a partial function f is the subset of X where it is defined, i.e., $\text{dom}(f) = \{x : f(x) \downarrow\}$.

Example 3.16. Every function $f: X \rightarrow Y$ is also a partial function. Partial functions that are defined everywhere on X —i.e., what we so far have simply called a function—are also called *total functions*.

Example 3.17. The partial function $f: \mathbb{R} \rightarrow \mathbb{R}$ given by $f(x) = 1/x$ is undefined for $x = 0$, and defined everywhere else.

3.7 Functions and Relations

A function which maps elements of X to elements of Y obviously defines a relation between X and Y , namely the relation which

holds between x and y iff $f(x) = y$. In fact, we might even—if we are interested in reducing the building blocks of mathematics for instance—*identify* the function f with this relation, i.e., with a set of pairs. This then raises the question: which relations define functions in this way?

Definition 3.18. Let $f: X \rightarrow Y$ be a partial function. The *graph* of f is the relation $R_f \subseteq X \times Y$ defined by

$$R_f = \{\langle x, y \rangle : f(x) = y\}.$$

Proposition 3.19. *Suppose $R \subseteq X \times Y$ has the property that whenever Rxy and Rxy' then $y = y'$. Then R is the graph of the partial function $f: X \rightarrow Y$ defined by: if there is a y such that Rxy , then $f(x) = y$, otherwise $f(x) \uparrow$. If R is also serial, i.e., for each $x \in X$ there is a $y \in Y$ such that Rxy , then f is total.*

Proof. Suppose there is a y such that Rxy . If there were another $y' \neq y$ such that Rxy' , the condition on R would be violated. Hence, if there is a y such that Rxy , that y is unique, and so f is well-defined. Obviously, $R_f = R$ and f is total if R is serial. \square

CHAPTER 4

The Size of Sets

4.1 Introduction

When Georg Cantor developed set theory in the 1870s, his interest was in part to make palatable the idea of an infinite collection—an actual infinity, as the medievals would say. Key to this rehabilitation of the notion of the infinite was a way to assign sizes—“cardinalities”—to sets. The cardinality of a finite set is just a natural number, e.g., \emptyset has cardinality 0, and a set containing five things has cardinality 5. But what about infinite sets? Do they all have the same cardinality, ∞ ? It turns out, they do not.

The first important idea here is that of an enumeration. We can list every finite set by listing all its elements. For some infinite sets, we can also list all their elements if we allow the list itself to be infinite. Such sets are called countable. Cantor’s surprising result was that some infinite sets are not countable.

4.2 Countable Sets

Definition 4.1. Informally, an *enumeration* of a set X is a list (possibly infinite) such that every element of X appears some

finite number of places into the list. If X has an enumeration, then X is said to be *countable*. If X is countable and infinite, we say X is countably infinite.

A couple of points about enumerations:

1. The order of elements of X in the enumeration does not matter, as long as every element appears: 4, 1, 25, 16, 9 enumerates the (set of the) first five square numbers just as well as 1, 4, 9, 16, 25 does.
2. Redundant enumerations are still enumerations: 1, 1, 2, 2, 3, 3, ... enumerates the same set as 1, 2, 3, ... does.
3. Order and redundancy *do* matter when we specify an enumeration: we can enumerate the natural numbers beginning with 1, 2, 3, 1, ..., but the pattern is easier to see when enumerated in the standard way as 1, 2, 3, 4, ...
4. Enumerations must have a beginning: ..., 3, 2, 1 is not an enumeration of the natural numbers because it has no first element. To see how this follows from the informal definition, ask yourself, “at what place in the list does the number 76 appear?”
5. The following is not an enumeration of the natural numbers: 1, 3, 5, ..., 2, 4, 6, ... The problem is that the even numbers occur at places $\infty + 1$, $\infty + 2$, $\infty + 3$, rather than at finite positions.
6. Lists may be gappy: 2, −, 4, −, 6, −, ... enumerates the even natural numbers.
7. The empty set is enumerable: it is enumerated by the empty list!

The following provides a more formal definition of an enumeration:

Definition 4.2. An *enumeration* of a set X is any surjective function $f : \mathbb{N} \rightarrow X$.

Let's convince ourselves that the formal definition and the informal definition using a possibly gappy, possibly infinite list are equivalent. A surjective function (partial or total) from \mathbb{N} to a set X enumerates X . Such a function determines an enumeration as defined informally above. Then an enumeration for X is the list $f(0), f(1), f(2), \dots$. Since f is surjective, every element of X is guaranteed to be the value of $f(n)$ for some $n \in \mathbb{N}$. Hence, every element of X appears at some finite place in the list. Since the function may be partial or not injective, the list may be gappy or redundant, but that is acceptable (as noted above). On the other hand, given a list that enumerates all elements of X , we can define a surjective function $f : \mathbb{N} \rightarrow X$ by letting $f(n)$ be the $(n+1)$ st member of the list, or undefined if the list has a gap in the $(n+1)$ st spot.

Example 4.3. A function enumerating the natural numbers (\mathbb{N}) is simply the identity function given by $f(n) = n$.

Example 4.4. The functions $f : \mathbb{N} \rightarrow \mathbb{N}$ and $g : \mathbb{N} \rightarrow \mathbb{N}$ given by

$$f(n) = 2n \text{ and} \tag{4.1}$$

$$g(n) = 2n + 1 \tag{4.2}$$

enumerate the even natural numbers and the odd natural numbers, respectively. However, neither function is an enumeration of \mathbb{N} , since neither is surjective.

Example 4.5. The function $f(n) = \lceil \frac{(-1)^n n}{2} \rceil$ (where $\lceil x \rceil$ denotes the *ceiling* function, which rounds x up to the nearest integer) enumerates the set of integers \mathbb{Z} . Notice how f generates the values of \mathbb{Z} by “hopping” back and forth between positive and

negative integers:

$$\begin{array}{cccccc}
 f(2) & f(3) & f(4) & f(5) & f(6) & \dots \\
 \lceil -\frac{1}{2} \rceil & \lceil \frac{2}{2} \rceil & \lceil -\frac{3}{2} \rceil & \lceil \frac{4}{2} \rceil & \lceil -\frac{5}{2} \rceil & \lceil \frac{6}{2} \rceil & \dots \\
 0 & 1 & -1 & 2 & -2 & 3 & \dots
 \end{array}$$

That is fine for “easy” sets. What about the set of, say, pairs of natural numbers?

$$\mathbb{N}^2 = \mathbb{N} \times \mathbb{N} = \{\langle n, m \rangle : n, m \in \mathbb{N}\}$$

Another method we can use to enumerate sets is to organize them in an *array*, such as the following:

	1	2	3	4	...
1	$\langle 1, 1 \rangle$	$\langle 1, 2 \rangle$	$\langle 1, 3 \rangle$	$\langle 1, 4 \rangle$...
2	$\langle 2, 1 \rangle$	$\langle 2, 2 \rangle$	$\langle 2, 3 \rangle$	$\langle 2, 4 \rangle$...
3	$\langle 3, 1 \rangle$	$\langle 3, 2 \rangle$	$\langle 3, 3 \rangle$	$\langle 3, 4 \rangle$...
4	$\langle 4, 1 \rangle$	$\langle 4, 2 \rangle$	$\langle 4, 3 \rangle$	$\langle 4, 4 \rangle$...
\vdots	\vdots	\vdots	\vdots	\vdots	\ddots

Clearly, every ordered pair in \mathbb{N}^2 will appear at least once in the array. In particular, $\langle n, m \rangle$ will appear in the n th column and m th row. But how do we organize the elements of an array into a list? The pattern in the array below demonstrates one way to do this:

	1	2	4	7	...
	3	5	8
	6	9
	10
	\vdots	\vdots	\vdots	\vdots	\ddots

This pattern is called *Cantor’s zig-zag method*. Other patterns are perfectly permissible, as long as they “zig-zag” through every cell

of the array. By Cantor's zig-zag method, the enumeration for \mathbb{N}^2 according to this scheme would be:

$$\langle 1, 1 \rangle, \langle 1, 2 \rangle, \langle 2, 1 \rangle, \langle 1, 3 \rangle, \langle 2, 2 \rangle, \langle 3, 1 \rangle, \langle 1, 4 \rangle, \langle 2, 3 \rangle, \langle 3, 2 \rangle, \langle 4, 1 \rangle, \dots$$

What ought we do about enumerating, say, the set of ordered triples of natural numbers?

$$\mathbb{N}^3 = \mathbb{N} \times \mathbb{N} \times \mathbb{N} = \{\langle n, m, k \rangle : n, m, k \in \mathbb{N}\}$$

We can think of \mathbb{N}^3 as the Cartesian product of \mathbb{N}^2 and \mathbb{N} , that is,

$$\mathbb{N}^3 = \mathbb{N}^2 \times \mathbb{N} = \{\langle \langle n, m \rangle, k \rangle : \langle n, m \rangle \in \mathbb{N}^2, k \in \mathbb{N}\}$$

and thus we can enumerate \mathbb{N}^3 with an array by labelling one axis with the enumeration of \mathbb{N} , and the other axis with the enumeration of \mathbb{N}^2 :

	1	2	3	4	...
$\langle 1, 1 \rangle$	$\langle 1, 1, 1 \rangle$	$\langle 1, 1, 2 \rangle$	$\langle 1, 1, 3 \rangle$	$\langle 1, 1, 4 \rangle$...
$\langle 1, 2 \rangle$	$\langle 1, 2, 1 \rangle$	$\langle 1, 2, 2 \rangle$	$\langle 1, 2, 3 \rangle$	$\langle 1, 2, 4 \rangle$...
$\langle 2, 1 \rangle$	$\langle 2, 1, 1 \rangle$	$\langle 2, 1, 2 \rangle$	$\langle 2, 1, 3 \rangle$	$\langle 2, 1, 4 \rangle$...
$\langle 1, 3 \rangle$	$\langle 1, 3, 1 \rangle$	$\langle 1, 3, 2 \rangle$	$\langle 1, 3, 3 \rangle$	$\langle 1, 3, 4 \rangle$...
\vdots	\vdots	\vdots	\vdots	\vdots	\ddots

Thus, by using a method like Cantor's zig-zag method, we may similarly obtain an enumeration of \mathbb{N}^3 .

4.3 Uncountable Sets

Some sets, such as the set \mathbb{N} of natural numbers, are infinite. So far we've seen examples of infinite sets which were all countable. However, there are also infinite sets which do not have this property. Such sets are called *uncountable*.

Cantor's method of diagonalization shows a set to be uncountable via a reductio proof. We start with the assumption that the set is countable, and show that a contradiction results from this assumption. Our first example is the set \mathbb{B}^ω of all infinite, non-gappy sequences of 0's and 1's.

Theorem 4.6. \mathbb{B}^ω is uncountable.

Proof. Suppose, for reductio, that \mathbb{B}^ω is countable, so that there is a list $s_1, s_2, s_3, s_4, \dots$ of all the elements of \mathbb{B}^ω . We may arrange this list, and the elements of each sequence s_i in it, in an array with the positive integers on the horizontal axis, as so:

	1	2	3	4	...
1	$s_1(1)$	$s_1(2)$	$s_1(3)$	$s_1(4)$...
2	$s_2(1)$	$s_2(2)$	$s_2(3)$	$s_2(4)$...
3	$s_3(1)$	$s_3(2)$	$s_3(3)$	$s_3(4)$...
4	$s_4(1)$	$s_4(2)$	$s_4(3)$	$s_4(4)$...
\vdots	\vdots	\vdots	\vdots	\vdots	\ddots

Here $s_1(1)$ is a name for whatever number, a 0 or a 1, is the first member in the sequence s_1 , and so on.

Now define \bar{s} as follows: The n th member $\bar{s}(n)$ of the sequence \bar{s} is set to

$$\bar{s}(n) = \begin{cases} 1 & \text{if } s_n(n) = 0 \\ 0 & \text{if } s_n(n) = 1. \end{cases}$$

In other words, $\bar{s}(n)$ has the opposite value to $s_n(n)$. To get $\bar{s}(n)$, take the 0's and 1's in the diagonal of the array, and swith every 0 to a 1 and every 1 to a 0.

Clearly \bar{s} is a non-gappy infinite sequence of 0s and 1s, since it is just the mirror sequence to the sequence of 0s and 1s that appear on the diagonal of our array. So \bar{s} is an element of \mathbb{B}^ω . Since it is an element of \mathbb{B}^ω , it must appear somewhere in the enumeration of \mathbb{B}^ω , that is, $\bar{s} = s_k$ for some k .

If $\bar{s} = s_k$, then for any m , $\bar{s}(m) = s_k(m)$. (This is just the criterion of identity for sequences—sequences are identical when they agree at every place.)

So in particular, $\bar{s}(k) = s_k(k)$. $\bar{s}(k)$ must be either an 0 or a 1. If it is a 0 then (given the definition of \bar{s}) $s_k(k)$ must be a 1. But if it is a 1 then $s_k(k)$ must be a 0. In either case $\bar{s}(k) \neq s_k(k)$. \square

This proof method is called “diagonalization” because it uses the diagonal of the array to define \bar{s} . Diagonalization need not involve the presence of an array: we can show that sets are not countable by using a similar idea even when no array and no actual diagonal is involved.

Theorem 4.7. $\wp(\mathbb{Z}^+)$ is not enumerable.

Proof. Suppose, for reductio, that $\wp(\mathbb{Z}^+)$ is countable, and so it has an enumeration, i.e., a list of all subsets of \mathbb{Z}^+ :

$$Z_1, Z_2, Z_3, \dots$$

We now define a set \bar{Z} such that for **any** positive integer i , $i \in \bar{Z}$ iff $i \notin Z_i$:

$$\bar{Z} = \{i \in \mathbb{Z}^+ : i \notin Z_i\}$$

\bar{Z} is clearly a set of positive integers, and thus $\bar{Z} \in \wp(\mathbb{Z}^+)$. So \bar{Z} must be $= Z_k$ for some $k \in \mathbb{Z}^+$. And if that is the case, i.e., $\bar{Z} = Z_k$, then $i \in \bar{Z}$ iff $i \in Z_k$ for all $i \in \mathbb{Z}^+$.

In particular, $k \in \bar{Z}$ iff $k \in Z_k$.

Now either $k \in Z_k$ or $k \notin Z_k$. In the first case, by the previous line, $k \in \bar{Z}$. But we’ve defined \bar{Z} so that it contains exactly those $i \in \mathbb{Z}^+$ which are *not* elements of Z_i . So by that definition, we would have to also have $k \notin Z_k$. In the second case, $k \notin Z_k$. But now k satisfies the condition by which we have defined \bar{Z} , and that means that $k \in \bar{Z}$. And as $\bar{Z} = Z_k$, we get that $k \in Z_k$ after all. Either case leads to a contradiction. \square

4.4 Reduction

We showed $\wp(\mathbb{Z}^+)$ to be uncountable by a diagonalization argument. However, with the proof that \mathbb{B}^ω , the set of all infinite sequences of 0s and 1s, is uncountable in place, we could have instead showed $\wp(\mathbb{Z}^+)$ to be uncountable by showing that *if $\wp(\mathbb{Z}^+)$ is countable then \mathbb{B}^ω is also countable*. This called *reducing* one problem to another.

Proof of Theorem 4.7 by reduction. Suppose, for reductio, that $\wp(\mathbb{Z}^+)$ is countable, and thus that there is an enumeration of it Z_1, Z_2, Z_3, \dots

Define the function $f: \wp(\mathbb{Z}^+) \rightarrow \mathbb{B}^\omega$ by letting $f(Z)$ be the sequence s_k such that $s_k(j) = 1$ iff $j \in Z$.

Every sequence of 0s and 1s corresponds to some set of positive integers, namely the one which has as its members those integers corresponding to the places where the sequence has 1s. In other words, this is a surjective function.

Now consider the list

$$f(Z_1), f(Z_2), f(Z_3), \dots$$

Since f is surjective, every member of \mathbb{B}^ω must appear as a value of f for some argument, and so must appear on the list. So this list must enumerate \mathbb{B}^ω .

So if $\wp(\mathbb{Z}^+)$ were countable, \mathbb{B}^ω would be countable. But \mathbb{B}^ω is uncountable (Theorem 4.6). \square

4.5 Equinumerous Sets

We have an intuitive notion of “size” of sets, which works fine for finite sets. But what about infinite sets? If we want to come up with a formal way of comparing the sizes of two sets of *any* size, it is a good idea to start with defining when sets are the same size. Let’s say sets of the same size are *equinumerous*. We want the formal notion of equinumerosity to correspond with our intuitive notion of “same size,” hence the formal notion ought to satisfy the following properties:

Reflexivity: Every set is equinumerous with itself.

Symmetry: For any sets X and Y , if X is equinumerous with Y , then Y is equinumerous with X .

Transitivity: For any sets X, Y , and Z , if X is equinumerous with Y and Y is equinumerous with Z , then X is equinumerous with Z .

In other words, we want equinumerosity to be an *equivalence relation*.

Definition 4.8. A set X is *equinumerous* with a set Y if and only if there is a total bijection f from X to Y (that is, $f: X \rightarrow Y$).

Proposition 4.9. *Equinumerosity defines an equivalence relation.*

Proof. Let X, Y , and Z be sets.

Reflexivity: Using the identity map $1_X: X \rightarrow X$, where $1_X(x) = x$ for all $x \in X$, we see that X is equinumerous with itself (clearly, 1_X is bijective).

Symmetry: Suppose that X is equinumerous with Y . Then there is a bijection $f: X \rightarrow Y$. Since f is bijective, its inverse f^{-1} is also a bijection. Since f is surjective, f^{-1} is total. Hence, $f^{-1}: Y \rightarrow X$ is a total bijection from Y to X , so Y is also equinumerous with X .

Transitivity: Suppose that X is equinumerous with Y via the total bijection f and that Y is equinumerous with Z via the total bijection g . Then the composition of $g \circ f: X \rightarrow Z$ is a total bijection, and X is thus equinumerous with Z .

Therefore, equinumerosity is an equivalence relation by the given definition. \square

Theorem 4.10. *Suppose X and Y are equinumerous. Then X is countable if and only if Y is.*

Proof. Let X and Y be equinumerous. Suppose that X is countable. Then there is a possibly partial, surjective function $f: \mathbb{N} \rightarrow X$. Since X and Y are equinumerous, there is a total bijection $g: X \rightarrow Y$. Claim: $g \circ f: \mathbb{N} \rightarrow Y$ is surjective. Clearly, $g \circ f$ is a function (since functions are closed under composition). To

see $g \circ f$ is surjective, let $y \in Y$. Since g is surjective, there is an $x \in X$ such that $g(x) = y$. Since f is surjective, there is an $n \in \mathbb{N}$ such that $f(n) = x$. Hence,

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

and thus $g \circ f$ is surjective. Since $g \circ f: \mathbb{N} \rightarrow Y$ is surjective, it is an enumeration of Y , and so Y is countable. \square

4.6 Comparing Sizes of Sets

Just like we were able to make precise when two sets have the same size in a way that also accounts for the size of infinite sets, we can also compare the sizes of sets in a precise way. Our definition of “is smaller than (or equinumerous)” will require, instead of a bijection between the sets, a total injective function from the first set to the second. If such a function exists, the size of the first set is less than or equal to the size of the second. Intuitively, an injective function from one set to another guarantees that the range of the function has at least as many elements as the domain, since no two elements of the domain map to the same element of the range.

Definition 4.11. $|X| \leq |Y|$ if and only if there is an injective function $f: X \rightarrow Y$.

Theorem 4.12 (Schröder-Bernstein). *Let X and Y be sets. If $|X| \leq |Y|$ and $|Y| \leq |X|$, then $|X| = |Y|$.*

In other words, if there is a total injective function from X to Y , and if there is a total injective function from Y back to X , then there is a total bijection from X to Y . Sometimes, it can be difficult to think of a bijection between two equinumerous sets, so the Schröder-Bernstein theorem allows us to break the comparison down into cases so we only have to think of an injection from the first to the second, and vice-versa. The Schröder-Bernstein

theorem, apart from being convenient, justifies the act of discussing the “sizes” of sets, for it tells us that set cardinalities have the familiar anti-symmetric property that numbers have.

Definition 4.13. $|X| < |Y|$ if and only if there is an injective function $f: X \rightarrow Y$ but no bijective $g: X \rightarrow Y$.

Theorem 4.14 (Cantor). *For all X , $|X| < |\wp(X)|$.*

Proof. The function $f: X \rightarrow \wp(X)$ that maps any $x \in X$ to its singleton $\{x\}$ is injective, since if $x \neq y$ then also $f(x) = \{x\} \neq \{y\} = f(y)$.

There cannot be a surjective function $g: X \rightarrow \wp(X)$, let alone a bijective one. For assume that a surjective $g: X \rightarrow \wp(X)$ exists. Then let $Y = \{x \in X : x \notin g(x)\}$. If $g(x)$ is defined for all $x \in X$, then Y is clearly a well-defined subset of X . If g is surjective, Y must be the value of g for some $x_0 \in X$, i.e., $Y = g(x_0)$. Now consider x_0 : it cannot be an element of Y , since if $x_0 \in Y$ then $x_0 \in g(x_0)$, and the definition of Y then would have $x_0 \notin Y$. On the other hand, it must be an element of Y , since if it were not, then $x_0 \notin Y = g(x_0)$. But then x_0 satisfies the defining condition of Y , and so $x_0 \in Y$. In either case, we have a contradiction. \square



Kurt Gödel

1906 - 1978

PART II

First-order Logic

CHAPTER 5

Syntax and Semantics

5.1 First-Order Languages

Expressions of first-order logic are built up from a basic vocabulary containing *variables*, *constant symbols*, *predicate symbols* and sometimes *function symbols*. From them, together with logical connectives, quantifiers, and punctuation symbols such as parentheses and commas, *terms* and *formulas* are formed.

Informally, predicate symbols are names for properties and relations, constant symbols are names for individual objects, and function symbols are names for mappings. These, except for the identity predicate $=$, are the *non-logical symbols* and together make up a language. Any first-order language \mathcal{L} is determined by its non-logical symbols. In the most general case, \mathcal{L} contains infinitely many symbols of each kind.

In the general case, we make use of the following symbols in first-order logic:

1. Logical symbols
 - a) Logical connectives: \neg (negation), \wedge (conjunction),

- \vee (disjunction), \rightarrow (conditional), \forall (universal quantifier), \exists (existential quantifier).
 - b) The propositional constant for falsity \perp .
 - c) The two-place identity predicate $=$.
 - d) A countably infinite set of variables: v_0, v_1, v_2, \dots
2. Non-logical symbols, making up the *standard language* of first-order logic
- a) A countably infinite set of n -place predicate symbols for each $n > 0$: $A_0^n, A_1^n, A_2^n, \dots$
 - b) A countably infinite set of constant symbols: c_0, c_1, c_2, \dots
 - c) A countably infinite set of n -place function symbols for each $n > 0$: $f_0^n, f_1^n, f_2^n, \dots$
3. Punctuation marks: $(,)$, and the comma.

Most of our definitions and results will be formulated for the full standard language of first-order logic. However, depending on the application, we may also restrict the language to only a few predicate symbols, constant symbols, and function symbols.

Example 5.1. The language \mathcal{L}_A of arithmetic contains a single two-place predicate symbol $<$, a single constant symbol 0 , one one-place function symbol ι , and two two-place function symbols $+$ and \times .

Example 5.2. The language of set theory \mathcal{L}_Z contains only the single two-place predicate symbol \in .

Example 5.3. The language of orders \mathcal{L}_{\leq} contains only the two-place predicate symbol \leq .

Again, these are conventions: officially, these are just aliases, e.g., $<$, \in , and \leq are aliases for A_0^2 , 0 for c_0 , ι for f_0^1 , $+$ for f_0^2 , \times for f_1^2 .

In addition to the primitive connectives and quantifiers introduced above, we also use the following *defined* symbols: \leftrightarrow (biconditional), truth \top

A defined symbol is not officially part of the language, but is introduced as an informal abbreviation: it allows us to abbreviate formulas which would, if we only used primitive symbols, get quite long. This is obviously an advantage. The bigger advantage, however, is that proofs become shorter. If a symbol is primitive, it has to be treated separately in proofs. The more primitive symbols, therefore, the longer our proofs.

You may be familiar with different terminology and symbols than the ones we use above. Logic texts (and teachers) commonly use either \sim , \neg , and $!$ for “negation”, \wedge , \cdot , and $\&$ for “conjunction”. Commonly used symbols for the “conditional” or “implication” are \rightarrow , \Rightarrow , and \supset . Symbols for “biconditional,” “bi-implication,” or “(material) equivalence” are \leftrightarrow , \Leftrightarrow , and \equiv . The \perp symbol is variously called “falsity,” “falsum,” “absurdity,” or “bottom.” The \top symbol is variously called “truth,” “verum,” or “top.”

It is conventional to use lower case letters (e.g., a , b , c) from the beginning of the Latin alphabet for constant symbols (sometimes called names), and lower case letters from the end (e.g., x , y , z) for variables. Quantifiers combine with variables, e.g., x ; notational variations include $\forall x$, $(\forall x)$, (x) , Πx , \bigwedge_x for the universal quantifier and $\exists x$, $(\exists x)$, (Ex) , Σx , \bigvee_x for the existential quantifier.

We might treat all the propositional operators and both quantifiers as primitive symbols of the language. We might instead choose a smaller stock of primitive symbols and treat the other logical operators as defined. “Truth functionally complete” sets of Boolean operators include $\{\neg, \vee\}$, $\{\neg, \wedge\}$, and $\{\neg, \rightarrow\}$ —these can be combined with either quantifier for an expressively complete first-order language.

You may be familiar with two other logical operators: the Sheffer stroke $|$ (named after Henry Sheffer), and Peirce’s arrow \downarrow , also known as Quine’s dagger. When given their usual readings of “nand” and “nor” (respectively), these operators are

truth functionally complete by themselves.

5.2 Terms and Formulas

Once a first-order language \mathcal{L} is given, we can define expressions built up from the basic vocabulary of \mathcal{L} . These include in particular *terms* and *formulas*.

Definition 5.4 (Terms). The set of *terms* $\text{Trm}(\mathcal{L})$ of \mathcal{L} is defined inductively by:

1. Every variable is a term.
2. Every constant symbol of \mathcal{L} is a term.
3. If f is an n -place function symbol and t_1, \dots, t_n are terms, then $f(t_1, \dots, t_n)$ is a term.
4. Nothing else is a term.

A term containing no variables is a *closed term*.

The constant symbols appear in our specification of the language and the terms as a separate category of symbols, but they could instead have been included as zero-place function symbols. We could then do without the second clause in the definition of terms. We just have to understand $f(t_1, \dots, t_n)$ as just f by itself if $n = 0$.

Definition 5.5 (Formula). The set of *formulas* $\text{Frm}(\mathcal{L})$ of the language \mathcal{L} is defined inductively as follows:

1. \perp is an atomic formula.
2. If R is an n -place predicate symbol of \mathcal{L} and t_1, \dots, t_n are terms of \mathcal{L} , then $R(t_1, \dots, t_n)$ is an atomic formula.
3. If t_1 and t_2 are terms of \mathcal{L} , then $=(t_1, t_2)$ is an atomic formula.

4. If A is a formula, then $\neg A$ is formula.
5. If A and B are formulas, then $(A \wedge B)$ is a formula.
6. If A and B are formulas, then $(A \vee B)$ is a formula.
7. If A and B are formulas, then $(A \rightarrow B)$ is a formula.
8. If A is a formula and x is a variable, then $\forall x A$ is a formula.
9. If A is a formula and x is a variable, then $\exists x A$ is a formula.
10. Nothing else is a formula.

The definitions of the set of terms and that of formulas are *inductive definitions*. Essentially, we construct the set of formulas in infinitely many stages. In the initial stage, we pronounce all atomic formulas to be formulas; this corresponds to the first few cases of the definition, i.e., the cases for \perp , $R(t_1, \dots, t_n)$ and $=(t_1, t_2)$. “Atomic formula” thus means any formula of this form.

The other cases of the definition give rules for constructing new formulas out of formulas already constructed. At the second stage, we can use them to construct formulas out of atomic formulas. At the third stage, we construct new formulas from the atomic formulas and those obtained in the second stage, and so on. A formula is anything that is eventually constructed at such a stage, and nothing else.

By convention, we write $=$ between its arguments and leave out the parentheses: $t_1 = t_2$ is an abbreviation for $=(t_1, t_2)$. Moreover, $\neg=(t_1, t_2)$ is abbreviated as $t_1 \neq t_2$. When writing a formula $(B * C)$ constructed from B, C using a two-place connective $*$, we will often leave out the outermost pair of parentheses and write simply $B * C$.

Some logic texts require that the variable x must occur in A in order for $\exists x A$ and $\forall x A$ to count as formulas. Nothing bad happens if you don’t require this, and it makes things easier.

Definition 5.6. Formulas constructed using the defined operators are to be understood as follows:

1. \top abbreviates $\neg\perp$.
2. $A \leftrightarrow B$ abbreviates $(A \rightarrow B) \wedge (B \rightarrow A)$.

If we work in a language for a specific application, we will often write two-place predicate symbols and function symbols between the respective terms, e.g., $t_1 < t_2$ and $(t_1 + t_2)$ in the language of arithmetic and $t_1 \in t_2$ in the language of set theory. The successor function in the language of arithmetic is even written conventionally *after* its argument: t' . Officially, however, these are just conventional abbreviations for $A_0^2(t_1, t_2)$, $f_0^2(t_1, t_2)$, $A_0^2(t_1, t_2)$ and $f_0^1(t)$, respectively.

Definition 5.7. The symbol \equiv expresses syntactic identity between strings of symbols, i.e., $A \equiv B$ iff A and B are strings of symbols of the same length and which contain the same symbol in each place.

The \equiv symbol may be flanked by strings obtained by concatenation, e.g., $A \equiv (B \vee C)$ means: the string of symbols A is the same string as the one obtained by concatenating an opening parenthesis, the string B , the \vee symbol, the string C , and a closing parenthesis, in this order. If this is the case, then we know that the first symbol of A is an opening parenthesis, A contains B as a substring (starting at the second symbol), that substring is followed by \vee , etc.

5.3 Unique Readability

The way we defined formulas guarantees that every formula has a *unique reading*, i.e., there is essentially only one way of constructing it according to our formation rules for formulas and only one way of “interpreting” it. If this were not so, we would have ambiguous formulas, i.e., formulas that have more than one reading or interpretation—and that is clearly something we want to avoid. But more importantly, without this property, most of the definitions and proofs we are going to give will not go through.

Perhaps the best way to make this clear is to see what would happen if we had given bad rules for forming formulas that would not guarantee unique readability. For instance, we could have forgotten the parentheses in the formation rules for connectives, e.g., we might have allowed this:

If A and B are formulas, then so is $A \rightarrow B$.

Starting from an atomic formula D , this would allow us to form $D \rightarrow D$, and from this, together with D , we would get $D \rightarrow D \rightarrow D$. But there are two ways to do this: one where we take D to be A and $D \rightarrow D$ to be B , and the other where A is $D \rightarrow D$ and B is D . Correspondingly, there are two ways to “read” the formula $D \rightarrow D \rightarrow D$. It is of the form $B \rightarrow C$ where B is D and C is $D \rightarrow D$, but *it is also* of the form $B \rightarrow C$ with B being $D \rightarrow D$ and C being D .

If this happens, our definitions will not always work. For instance, when we define the main operator of a formula, we say: in a formula of the form $B \rightarrow C$, the main operator is the indicated occurrence of \rightarrow . But if we can match the formula $D \rightarrow D \rightarrow D$ with $B \rightarrow C$ in the two different ways mentioned above, then in one case we get the first occurrence of \rightarrow as the main operator, and in the second case the second occurrence. But we intend the main operator to be a *function* of the formula, i.e., every formula must have exactly one main operator occurrence.

Lemma 5.8. *The number of left and right parentheses in a formula A are equal.*

Proof. We prove this by induction on the way A is constructed. This requires two things: (a) We have to prove first that all atomic formulas have the property in question (the induction basis). (b) Then we have to prove that when we construct new formulas out of given formulas, the new formulas have the property provided the old ones do.

Let $l(A)$ be the number of left parentheses, and $r(A)$ the number of right parentheses in A , and $l(t)$ and $r(t)$ similarly the number of left and right parentheses in a term t . We leave the proof that for any term t , $l(t) = r(t)$ as an exercise.

1. $A \equiv \perp$: A has 0 left and 0 right parentheses.
2. $A \equiv R(t_1, \dots, t_n)$: $l(A) = 1 + l(t_1) + \dots + l(t_n) = 1 + r(t_1) + \dots + r(t_n) = r(A)$. Here we make use of the fact, left as an exercise, that $l(t) = r(t)$ for any term t .
3. $A \equiv t_1 = t_2$: $l(A) = l(t_1) + l(t_2) = r(t_1) + r(t_2) = r(A)$.
4. $A \equiv \neg B$: By induction hypothesis, $l(B) = r(B)$. Thus $l(A) = l(B) = r(B) = r(A)$.
5. $A \equiv (B * C)$: By induction hypothesis, $l(B) = r(B)$ and $l(C) = r(C)$. Thus $l(A) = 1 + l(B) + l(C) = 1 + r(B) + r(C) = r(A)$.
6. $A \equiv \forall x B$: By induction hypothesis, $l(B) = r(B)$. Thus, $l(A) = l(B) = r(B) = r(A)$.
7. $A \equiv \exists x B$: Similarly.

□

Definition 5.9. A string of symbols B is a proper prefix of a string of symbols A if concatenating B and a non-empty string of symbols yields A .

Lemma 5.10. *If A is a formula, and B is a proper prefix of A , then B is not a formula.*

Proof. Exercise.

□

Proposition 5.11. *If A is an atomic formula, then it satisfies one, and only one of the following conditions.*

1. $A \equiv \perp$.
2. $A \equiv R(t_1, \dots, t_n)$ where R is an n -place predicate symbol, t_1, \dots, t_n are terms, and each of R, t_1, \dots, t_n is uniquely determined.
3. $A \equiv t_1 = t_2$ where t_1 and t_2 are uniquely determined terms.

Proof. Exercise. □

Proposition 5.12 (Unique Readability). *Every formula satisfies one, and only one of the following conditions.*

1. A is atomic.
2. A is of the form $\neg B$.
3. A is of the form $(B \wedge C)$.
4. A is of the form $(B \vee C)$.
5. A is of the form $(B \rightarrow C)$.
6. A is of the form $\forall x B$.
7. A is of the form $\exists x B$.

Moreover, in each case B , or B and C , are uniquely determined. This means that, e.g., there are no different pairs B, C and B', C' so that A is both of the form $(B \rightarrow C)$ and $(B' \rightarrow C')$.

Proof. The formation rules require that if a formula is not atomic, it must start with an opening parenthesis $($, \neg , or with a quantifier. On the other hand, every formula that start with one of the following symbols must be atomic: a predicate symbol, a function symbol, a constant symbol, \perp .

So we really only have to show that if A is of the form $(B * C)$ and also of the form $(B' *' C')$, then $B \equiv B'$, $C \equiv C'$, and $* = *'$.

So suppose both $A \equiv (B * C)$ and $A \equiv (B' *' C')$. Then either $B \equiv B'$ or not. If it is, clearly $* = *'$ and $C \equiv C'$, since they then are substrings of A that begin in the same place and are of the same length. The other case is $C \not\equiv C'$. Since C and C' are both substrings of A that begin at the same place, one must be a prefix of the other. But this is impossible by [Lemma 5.10](#). \square

5.4 Main operator of a Formula

It is often useful to talk about the last operator used in constructing a formula A . This operator is called the *main operator* of A . Intuitively, it is the “outermost” operator of A . For example, the main operator of $\neg A$ is \neg , the main operator of $(A \vee B)$ is \vee , etc.

Definition 5.13 (Main operator). The *main operator* of a formula A is defined as follows:

1. A is atomic: A has no main operator.
2. $A \equiv \neg B$: the main operator of A is \neg .
3. $A \equiv (B \wedge C)$: the main operator of A is \wedge .
4. $A \equiv (B \vee C)$: the main operator of A is \vee .
5. $A \equiv (B \rightarrow C)$: the main operator of A is \rightarrow .
6. $A \equiv \forall x B$: the main operator of A is \forall .
7. $A \equiv \exists x B$: the main operator of A is \exists .

In each case, we intend the specific indicated *occurrence* of the main operator in the formula. For instance, since the formula $((D \rightarrow E) \rightarrow (E \rightarrow D))$ is of the form $(B \rightarrow C)$ where B is $(D \rightarrow E)$ and C is $(E \rightarrow D)$, the second occurrence of \rightarrow is the main operator.

This is a *recursive* definition of a function which maps all non-atomic formulas to their main operator occurrence. Because of the way formulas are defined inductively, every formula A satisfies one of the cases in [Definition 5.13](#). This guarantees that for each non-atomic formula A a main operator exists. Because each formula satisfies only one of these conditions, and because the smaller formulas from which A is constructed are uniquely determined in each case, the main operator occurrence of A is unique, and so we have defined a function.

We call formulas by the following names depending on which symbol their main operator is:

Main operator	Type of formula	Example
none	atomic (formula)	$\perp, R(t_1, \dots, t_n), t_1 = t_2$
\neg	negation	$\neg A$
\wedge	conjunction	$(A \wedge B)$
\vee	disjunction	$(A \vee B)$
\rightarrow	conditional	$(A \rightarrow B)$
\forall	universal (formula)	$\forall x A$
\exists	existential (formula)	$\exists x A$

5.5 Subformulas

It is often useful to talk about the formulas that “make up” a given formula. We call these its *subformulas*. Any formula counts as a subformula of itself; a subformula of A other than A itself is a *proper subformula*.

Definition 5.14 (Immediate Subformula). If A is a formula, the *immediate subformulas* of A are defined inductively as follows:

1. Atomic formulas have no immediate subformulas.
2. $A \equiv \neg B$: The only immediate subformula of A is B .
3. $A \equiv (B * C)$: The immediate subformulas of A are B and C ($*$ is any one of the two-place connectives).

4. $A \equiv \forall x B$: The only immediate subformula of A is B .
5. $A \equiv \exists x B$: The only immediate subformula of A is B .

Definition 5.15 (Proper Subformula). If A is a formula, the *proper subformulas* of A are recursively as follows:

1. Atomic formulas have no proper subformulas.
2. $A \equiv \neg B$: The proper subformulas of A are B together with all proper subformulas of B .
3. $A \equiv (B * C)$: The proper subformulas of A are B , C , together with all proper subformulas of B and those of C .
4. $A \equiv \forall x B$: The proper subformulas of A are B together with all proper subformulas of B .
5. $A \equiv \exists x B$: The proper subformulas of A are B together with all proper subformulas of B .

Definition 5.16 (Subformula). The subformulas of A are A itself together with all its proper subformulas.

Note the subtle difference in how we have defined immediate subformulas and proper subformulas. In the first case, we have directly defined the immediate subformulas of a formula A for each possible form of A . It is an explicit definition by cases, and the cases mirror the inductive definition of the set of formulas. In the second case, we have also mirrored the way the set of all formulas is defined, but in each case we have also included the proper subformulas of the smaller formulas B , C in addition to these formulas themselves. This makes the definition *recursive*. In general, a definition of a function on an inductively defined set (in our case, formulas) is recursive if the cases in the definition of the function make use of the function itself. To be well defined, we must make sure, however, that we only ever use the values of the function for arguments that come “before” the one we are defining—in our case, when defining “proper subformula” for $(B*$

C) we only use the proper subformulas of the “earlier” formulas B and C .

5.6 Free Variables and Sentences

Definition 5.17 (Free occurrences of a variable). The *free* occurrences of a variable in a formula are defined inductively as follows:

1. A is atomic: all variable occurrences in A are free.
2. $A \equiv \neg B$: the free variable occurrences of A are exactly those of B .
3. $A \equiv (B * C)$: the free variable occurrences of A are those in B together with those in C .
4. $A \equiv \forall x B$: the free variable occurrences in A are all of those in B except for occurrences of x .
5. $A \equiv \exists x B$: the free variable occurrences in A are all of those in B except for occurrences of x .

Definition 5.18 (Bound Variables). An occurrence of a variable in a formula A is *bound* if it is not free.

Definition 5.19 (Scope). If $\forall x B$ is an occurrence of a subformula in a formula A , then the corresponding occurrence of B in A is called the *scope* of the corresponding occurrence of $\forall x$. Similarly for $\exists x$.

If B is the scope of a quantifier occurrence $\forall x$ or $\exists x$ in A , then all occurrences of x which are free in B are said to be *bound* by the mentioned quantifier occurrence.

Example 5.20. Here is a somewhat complicated formula A :

$$\forall x_0 \underbrace{(A_0^1(x_0) \rightarrow A_0^2(x_0, x_1))}_B \rightarrow \exists x_1 \underbrace{(A_1^2(x_0, x_1) \vee \forall x_0 \overbrace{\neg A_1^1(x_0)}^D)}_C$$

B is the scope of the first $\forall x_0$, C is the scope of $\exists x_1$, and D is the scope of the second $\forall x_0$. The first $\forall x_0$ binds the occurrences of x_0 in B , $\exists x_1$ the occurrence of x_1 in C , and the second $\forall x_0$ binds the occurrence of x_0 in D . The first occurrence of x_1 and the fourth occurrence of x_0 are free in A . The last occurrence of x_0 is free in D , but bound in C and A .

Definition 5.21 (Sentence). A formula A is a *sentence* iff it contains no free occurrences of variables.

5.7 Substitution

Definition 5.22 (Substitution in a term). We define $s[t/x]$, the result of *substituting* t for every occurrence of x in s , recursively:

1. $s \equiv c$: $s[t/x]$ is just s .
2. $s \equiv y$: $s[t/x]$ is also just s , provided y is a variable other than x .
3. $s \equiv x$: $s[t/x]$ is t .
4. $s \equiv f(t_1, \dots, t_n)$: $s[t/x]$ is $f(t_1[t/x], \dots, t_n[t/x])$.

Definition 5.23. A term t is *free for* x in A if none of the free occurrences of x in A occur in the scope of a quantifier that binds a variable in t .

Definition 5.24 (Substitution in a formula). If A is a formula, x is a variable, and t is a term free for x in A , then $A[t/x]$ is the result of substituting t for all free occurrences of x in A .

1. $A \equiv P(t_1, \dots, t_n)$: $A[t/x]$ is $P(t_1[t/x], \dots, t_n[t/x])$.
2. $A \equiv t_1 = t_2$: $A[t/x]$ is $t_1[t/x] = t_2[t/x]$.
3. $A \equiv \neg B$: $A[t/x]$ is $\neg B[t/x]$.
4. $A \equiv (B \wedge C)$: $A[t/x]$ is $(B[t/x] \wedge C[t/x])$.

5. $A \equiv (B \vee C)$: $A[t/x]$ is $(B[t/x] \vee C[t/x])$.
6. $A \equiv (B \rightarrow C)$: $A[t/x]$ is $(B[t/x] \rightarrow C[t/x])$.
7. $A \equiv \forall y B$: $A[t/x]$ is $\forall y B[t/x]$, provided y is a variable other than x ; otherwise $A[t/x]$ is just A .
8. $A \equiv \exists y B$: $A[t/x]$ is $\exists y B[t/x]$, provided y is a variable other than x ; otherwise $A[t/x]$ is just A .

Note that substitution may be vacuous: If x does not occur in A at all, then $A[t/x]$ is just A .

The restriction that t must be free for x in A is necessary to exclude cases like the following. If $A \equiv \exists y x < y$ and $t \equiv y$, then $A[t/y]$ would be $\exists y y < y$. In this case the free variable y is “captured” by the quantifier $\exists y$ upon substitution, and that is undesirable. For instance, we would like it to be the case that whenever $\forall x B$ holds, so does $B[t/x]$. But consider $\forall x \exists y x < y$ (here B is $\exists y x < y$). It is sentence that is true about, e.g., the natural numbers: for every number x there is a number y greater than it. If we allowed y as a possible substitution for x , we would end up with $B[y/x] \equiv \exists y y < y$, which is false. We prevent this by requiring that none of the free variables in t would end up being bound by a quantifier in A .

We often use the following convention to avoid cumbersome notation: If A is a formula with a free variable x , we write $A(x)$ to indicate this. When it is clear which A and x we have in mind, and t is a term (assumed to be free for x in $A(x)$), then we write $A(t)$ as short for $A(x)[t/x]$.

5.8 Structures for First-order Languages

First-order languages are, by themselves, *uninterpreted*: the constant symbols, function symbols, and predicate symbols have no specific meaning attached to them. Meanings are given by specifying a *structure*. It specifies the *domain*, i.e., the objects which the constant symbols pick out, the function symbols operate on,

and the quantifiers range over. In addition, it specifies which constant symbols pick out which objects, how a function symbol maps objects to objects, and which objects the predicate symbols apply to. Structures are the basis for *semantic* notions in logic, e.g., the notion of consequence, validity, satisfiability. They are variously called “structures,” “interpretations,” or “models” in the literature.

Definition 5.25 (Structures). A *structure* M , for a language \mathcal{L} of first-order logic consists of the following elements:

1. *Domain*: a non-empty set, $|M|$
2. *Interpretation of constant symbols*: for each constant symbol c of \mathcal{L} , an element $c^M \in |M|$
3. *Interpretation of predicate symbols*: for each n -place predicate symbol R of \mathcal{L} (other than $=$), an n -ary relation $R^M \subseteq |M|^n$
4. *Interpretation of function symbols*: for each n -place function symbol f of \mathcal{L} , an n -ary function $f^M: |M|^n \rightarrow |M|$

Example 5.26. A structure M for the language of arithmetic consists of a set, an element of $|M|$, 0^M , as interpretation of the constant symbol 0 , a one-place function $\iota^M: |M| \rightarrow |M|$, two two-place functions $+^M$ and \times^M , both $|M|^2 \rightarrow |M|$, and a two-place relation $<^M \subseteq |M|^2$.

An obvious example of such a structure is the following:

1. $|N| = \mathbb{N}$
2. $0^N = 0$
3. $\iota^N(n) = n + 1$ for all $n \in \mathbb{N}$
4. $+^N(n, m) = n + m$ for all $n, m \in \mathbb{N}$
5. $\times^N(n, m) = n \cdot m$ for all $n, m \in \mathbb{N}$
6. $<^N = \{\langle n, m \rangle : n \in \mathbb{N}, m \in \mathbb{N}, n < m\}$

The structure N for \mathcal{L}_A so defined is called the *standard model of arithmetic*, because it interprets the non-logical constants of \mathcal{L}_A exactly how you would expect.

However, there are many other possible structures for \mathcal{L}_A . For instance, we might take as the domain the set \mathbb{Z} of integers instead of \mathbb{N} , and define the interpretations of 0 , 1 , $+$, \times , $<$ accordingly. But we can also define structures for \mathcal{L}_A which have nothing even remotely to do with numbers.

Example 5.27. A structure M for the language \mathcal{L}_Z of set theory requires just a set and a single-two place relation. So technically, e.g., the set of people plus the relation “ x is older than y ” could be used as a structure for \mathcal{L}_Z , as well as \mathbb{N} together with $n \geq m$ for $n, m \in \mathbb{N}$.

A particularly interesting structure for \mathcal{L}_Z in which the elements of the domain are actually sets, and the interpretation of \in actually is the relation “ x is an element of y ” is the structure HF of *hereditarily finite sets*:

1. $|HF| = \emptyset \cup \wp(\emptyset) \cup \wp(\wp(\emptyset)) \cup \wp(\wp(\wp(\emptyset))) \cup \dots$;
2. $\in^{HF} = \{\langle x, y \rangle : x, y \in |HF|, x \in y\}$.

Recall that a term is *closed* if it contains no variables.

Definition 5.28 (Value of closed terms). If t is a closed term of the language \mathcal{L} and M is a structure for \mathcal{L} , the *value* $\text{Val}^M(t)$ is defined as follows:

1. If t is just the constant symbol c , then $\text{Val}^M(c) = c^M$.
2. If t is of the form $f(t_1, \dots, t_n)$, then

$$\text{Val}^M(t) = f^M(\text{Val}^M(t_1), \dots, \text{Val}^M(t_n)).$$

Definition 5.29 (Covered structure). A structure is *covered* if every element of the domain is the value of some closed term.

Example 5.30. Let \mathcal{L} be the language with constant symbols *zero*, *one*, *two*, \dots , the binary predicate symbols $=$ and $<$, and the binary function symbols $+$ and \times . Then a structure M for \mathcal{L} is the one with domain $|M| = \{0, 1, 2, \dots\}$ and name assignment $zero^M = 0$, $one^M = 1$, $two^M = 2$, and so forth. For the binary relation symbol $<$, the set $<^M$ is the set of all pairs $\langle c_1, c_2 \rangle \in |M|^2$ such that the integer c_1 is less than the integer c_2 : for example, $\langle 1, 3 \rangle \in <^M$ but $\langle 2, 2 \rangle \notin <^M$. For the binary function symbol $+$, define $+^M$ in the usual way—for example, $+^M(2, 3)$ maps to 5, and similarly for the binary function symbol \times . Hence, the value of *four* is just 4, and the value of $\times(two, +(three, zero))$ (or in infix notation, $two \times (three + zero)$) is

$$\begin{aligned}
 \text{Val}^M(\times(two, +(three, zero))) &= \\
 &= \times^M(\text{Val}^M(two), \text{Val}^M(two, +(three, zero))) \\
 &= \times^M(\text{Val}^M(two), +^M(\text{Val}^M(three), \text{Val}^M(zero))) \\
 &= \times^M(two^M, +^M(three^M, zero^M)) \\
 &= \times^M(2, +^M(3, 0)) \\
 &= \times^M(2, 3) \\
 &= 6
 \end{aligned}$$

The stipulations we make as to what counts as a structure impact our logic. For example, the choice to prevent empty domains ensures, given the usual account of satisfaction (or truth) for quantified sentences, that $\exists x (A(x) \vee \neg A(x))$ is valid—that is, a logical truth. And the stipulation that all constant symbols must refer to an object in the domain ensures that the existential generalization is a sound pattern of inference: $A(a)$, therefore $\exists x A(x)$. If we allowed names to refer outside the domain, or to not refer, then we would be on our way to a *free logic*, in which existential generalization requires an additional premise: $A(a)$ and $\exists x x = a$, therefore $\exists x A(x)$.

5.9 Satisfaction of a Formula in a Structure

The basic notion that relates expressions such as terms and formulas, on the one hand, and structures on the other, are those of *value* of a term and *satisfaction* of a formula. Informally, the value of a term is an element of a structure—if the term is just a constant, its value is the object assigned to the constant by the structure, and if it is built up using function symbols, the value is computed from the values of constants and the functions assigned to the functions in the term. A formula is *satisfied* in a structure if the interpretation given to the predicates makes the formula true in the domain of the structure. This notion of satisfaction is specified inductively: the specification of the structure directly states when atomic formulas are satisfied, and we define when a complex formula is satisfied depending on the main connective or quantifier and whether or not the immediate subformulas are satisfied. The case of the quantifiers here is a bit tricky, as the immediate subformula of a quantified formula has a free variable, and structures don't specify the values of variables. In order to deal with this difficulty, we also introduce *variable assignments* and define satisfaction not with respect to a structure alone, but with respect to a structure plus a variable assignment.

Definition 5.31 (Variable Assignment). A *variable assignment* s for a structure M is a function which maps each variable to an element of $|M|$, i.e., $s: \text{Var} \rightarrow |M|$.

A structure assigns a value to each constant symbol, and a variable assignment to each variable. But we want to use terms built up from them to also name elements of the domain. For this we define the value of terms inductively. For constant symbols and variables the value is just as the structure or the variable assignment specifies it; for more complex terms it is computed recursively using the functions the structure assigns to the function symbols.

Definition 5.32 (Value of Terms). If t is a term of the language \mathcal{L} , M is a structure for \mathcal{L} , and s is a variable assignment for M , the *value* $\text{Val}_s^M(t)$ is defined as follows:

1. $t \equiv c$: $\text{Val}_s^M(t) = c^M$.
2. $t \equiv x$: $\text{Val}_s^M(t) = s(x)$.
3. $t \equiv f(t_1, \dots, t_n)$:

$$\text{Val}_s^M(t) = f^M(\text{Val}_s^M(t_1), \dots, \text{Val}_s^M(t_n)).$$

Definition 5.33 (x -Variant). If s is a variable assignment for a structure M , then any variable assignment s' for M which differs from s at most in what it assigns to x is called an x -variant of s . If s' is an x -variant of s we write $s \sim_x s'$.

Note that an x -variant of an assignment s does not *have* to assign something different to x . In fact, every assignment counts as an x -variant of itself.

Definition 5.34 (Satisfaction). Satisfaction of a formula A in a structure M relative to a variable assignment s , in symbols: $M, s \models A$, is defined recursively as follows. (We write $M, s \not\models A$ to mean “not $M, s \models A$.”)

1. $A \equiv \perp$: not $M, s \models A$.
2. $A \equiv R(t_1, \dots, t_n)$: $M, s \models A$ iff $\langle \text{Val}_s^M(t_1), \dots, \text{Val}_s^M(t_n) \rangle \in R^M$.
3. $A \equiv t_1 = t_2$: $M, s \models A$ iff $\text{Val}_s^M(t_1) = \text{Val}_s^M(t_2)$.
4. $A \equiv \neg B$: $M, s \models A$ iff $M, s \not\models B$.
5. $A \equiv (B \wedge C)$: $M, s \models A$ iff $M, s \models B$ and $M, s \models C$.
6. $A \equiv (B \vee C)$: $M, s \models A$ iff $M, s \models B$ or $M, s \models C$ (or both).
7. $A \equiv (B \rightarrow C)$: $M, s \models A$ iff $M, s \not\models B$ or $M, s \models C$ (or both).

8. $A \equiv \forall x B$: $M, s \models A$ iff for every x -variant s' of s , $M, s' \models B$.
9. $A \equiv \exists x B$: $M, s \models A$ iff there is an x -variant s' of s so that $M, s' \models B$.

The variable assignments are important in the last two clauses. We cannot define satisfaction of $\forall x B(x)$ by “for all $a \in |M|$, $M \models B(a)$.” We cannot define satisfaction of $\exists x B(x)$ by “for at least one $a \in |M|$, $M \models B(a)$.” The reason is that a is not symbol of the language, and so $B(a)$ is not a formula (that is, $B[a/x]$ is undefined). We also cannot assume that we have constant symbols or terms available that name every element of M , since there is nothing in the definition of structures that requires it. Even in the standard language the set of constant symbols is countably infinite, so if $|M|$ is not countable there aren’t even enough constant symbols to name every object.

A variable assignment s provides a value for *every* variable in the language. This is of course not necessary: whether or not a formula A is satisfied in a structure with respect to s only depends on the assignments s makes to the free variables that actually occur in A . This is the content of the next theorem. We require variable assignments to assign values to all variables simply because it makes things a lot easier.

Proposition 5.35. *If x_1, \dots, x_n are the only free variables in A and $s(x_i) = s'(x_i)$ for $i = 1, \dots, n$, then $M, s \models A$ iff $M, s' \models A$.*

Proof. We use induction on the complexity of A . For the base case, where A is atomic, A can be: \perp , $R(t_1, \dots, t_k)$ for a k -place predicate R and terms t_1, \dots, t_k , or $t_1 = t_2$ for terms t_1 and t_2 .

1. $A \equiv \perp$: both $M, s \not\models A$ and $M, s' \not\models A$.
2. $A \equiv R(t_1, \dots, t_k)$: let $M, s \models A$. Then

$$\langle \text{Val}_s^M(t_1), \dots, \text{Val}_s^M(t_k) \rangle \in R^M.$$

For $i = 1, \dots, k$, if t_i is a constant, then $\text{Val}_s^M(t_i) = \text{Val}^M(t_i) = \text{Val}_{s'}^M(t_i)$. If t_i is a free variable, then since the mappings s and s' agree on all free variables, $\text{Val}_s^M(t_i) = s(t_i) = s'(t_i) = \text{Val}_{s'}^M(t_i)$. Similarly, if t_i is of the form $f(t'_1, \dots, t'_j)$, we will also get $\text{Val}_s^M(t_i) = \text{Val}_{s'}^M(t_i)$. Hence, $\text{Val}_s^M(t_i) = \text{Val}_{s'}^M(t_i)$ for any term t_i for $i = 1, \dots, k$, so we also have $\langle \text{Val}_s^M(t_1), \dots, \text{Val}_{s'}^M(t_k) \rangle \in R^M$.

3. $A \equiv t_1 = t_2$: if $M, s \models A$, $\text{Val}_{s'}^M(t_1) = \text{Val}_s^M(t_1) = \text{Val}_s^M(t_2) = \text{Val}_{s'}^M(t_2)$, so $M, s' \models t_1 = t_2$.

Now assume $M, s \models B$ iff $M, s' \models B$ for all formulas B less complex than A . The induction step proceeds by cases determined by the main operator of A . In each case, we only demonstrate the forward direction of the biconditional; the proof of the reverse direction is symmetrical.

1. $A \equiv \neg B$: if $M, s \models A$, then $M, s \not\models B$, so by the induction hypothesis, $M, s' \not\models B$, hence $M, s' \models A$.
2. $A \equiv B \wedge C$: exercise.
3. $A \equiv B \vee C$: if $M, s \models A$, then $M, s \models B$ or $M, s \models C$. By induction hypothesis, $M, s' \models B$ or $M, s' \models C$, so $M, s' \models A$.
4. $A \equiv B \rightarrow C$: exercise.
5. $A \equiv \exists x B$: if $M, s \models A$, there is an x -variant \bar{s} of s so that $M, \bar{s} \models B$. Let \bar{s}' denote the x -variant of s' that assigns the same thing to x as does \bar{s} : then by the induction hypothesis, $M, \bar{s}' \models B$. Hence, there is an x -variant of s' that satisfies B , so $M, s' \models A$.
6. $A \equiv \forall x B$: exercise.

By induction, we get that $M, s \models A$ iff $M, s' \models A$ whenever x_1, \dots, x_n are the only free variables in A and $s(x_i) = s'(x_i)$ for $i = 1, \dots, n$. □

Definition 5.36. If A is a sentence, we say that a structure M *satisfies* A , $M \models A$, iff $M, s \models A$ for all variable assignments s .

If $M \models A$, we also say that A is *true in* M .

Proposition 5.37. Suppose $A(x)$ only contains x free, and M is a structure. Then:

1. $M \models \exists x A(x)$ iff $M, s \models A(x)$ for at least one variable assignment s .
2. $M \models \forall x A(x)$ iff $M, s \models A(x)$ for all variable assignments s .

Proof. Exercise. □

5.10 Extensionality

Extensionality, sometimes called relevance, can be expressed informally as follows: the only thing that bears upon the satisfaction of formula A in a structure M relative to a variable assignment s , are the assignments made by M and s to the elements of the language that actually appear in A .

One immediate consequence of extensionality is that where two structures M and M' agree on all the elements of the language appearing in a sentence A and have the same domain, M and M' must also agree on A itself.

Proposition 5.38 (Extensionality). Let A be a sentence, and M and M' be structures. If $c^M = c^{M'}$, $R^M = R^{M'}$, and $f^M = f^{M'}$ for every constant symbol c , relation symbol R , and function symbol f occurring in A , then $M \models A$ iff $M' \models A$.

Moreover, the value of a term, and whether or not a structure satisfies a formula, only depends on the values of its subterms.

Proposition 5.39. *Let M be a structure, t and t' terms, and s a variable assignment. Let $s' \sim_x s$ be the x -variant of s given by $s'(x) = \text{Val}_s^M(t')$. Then $\text{Val}_s^M(t[t'/x]) = \text{Val}_{s'}^M(t)$.*

Proof. By induction on t .

1. If t is a constant, say, $t \equiv c$, then $t[t'/x] = c$, and $\text{Val}_s^M(c) = c^M = \text{Val}_{s'}^M(c)$.
2. If t is a variable other than x , say, $t \equiv y$, then $t[t'/x] = y$, and $\text{Val}_s^M(y) = \text{Val}_{s'}^M(y)$ since $s' \sim_x s$.
3. If $t \equiv x$, then $t[t'/x] = t'$. But $\text{Val}_s^M(x) = \text{Val}_s^M(t')$ by definition of s' .
4. If $t \equiv f(t_1, \dots, t_n)$ then we have:

$$\begin{aligned}
 \text{Val}_s^M(t[t'/x]) &= \\
 &= \text{Val}_s^M(f(t_1[t'/x], \dots, t_n[t'/x])) \\
 &\quad \text{by definition of } t[t'/x] \\
 &= f^M(\text{Val}_s^M(t_1[t'/x]), \dots, \text{Val}_s^M(t_n[t'/x])) \\
 &\quad \text{by definition of } \text{Val}_s^M(f(\dots)) \\
 &= f^M(\text{Val}_{s'}^M(t_1), \dots, \text{Val}_{s'}^M(t_n)) \\
 &\quad \text{by induction hypothesis} \\
 &= \text{Val}_{s'}^M(t) \text{ by definition of } \text{Val}_{s'}^M(f(\dots))
 \end{aligned}$$

□

Proposition 5.40. *Let M be a structure, A a formula, t a term, and s a variable assignment. Let $s' \sim_x s$ be the x -variant of s given by $s'(x) = \text{Val}_s^M(t)$. Then $M, s \models A[t/x]$ iff $M, s' \models A$.*

Proof. Exercise. □

5.11 Semantic Notions

Give the definition of structures for first-order languages, we can define some basic semantic properties of and relationships between sentences. The simplest of these is the notion of *validity* of a sentence. A sentence is valid if it is satisfied in every structure. Valid sentences are those that are satisfied regardless of how the non-logical symbols in it are interpreted. Valid sentences are therefore also called *logical truths*—they are true, i.e., satisfied, in any structure and hence their truth depends only on the logical symbols occurring in them and their syntactic structure, but not on the non-logical symbols or their interpretation.

Definition 5.41 (Validity). A sentence A is *valid*, $\models A$, iff $M \models A$ for every structure M .

Definition 5.42 (Entailment). A set of sentences Γ *entails* a sentence A , $\Gamma \models A$, iff for every structure M with $M \models \Gamma$, $M \models A$.

Definition 5.43 (Satisfiability). A set of sentences Γ is *satisfiable* if $M \models \Gamma$ for some structure M . If Γ is not satisfiable it is called *unsatisfiable*.

Proposition 5.44. *A sentence A is valid iff $\Gamma \models A$ for every set of sentences Γ .*

Proof. For the forward direction, let A be valid, and let Γ be a set of sentences. Let M be a structure so that $M \models \Gamma$. Since A is valid, $M \models A$, hence $\Gamma \models A$.

For the contrapositive of the reverse direction, let A be invalid, so there is a structure M with $M \not\models A$. When $\Gamma = \{\top\}$, since \top is valid, $M \models \Gamma$. Hence, there is a structure M so that $M \models \Gamma$ but $M \not\models A$, hence Γ does not entail A . \square

Proposition 5.45. *$\Gamma \models A$ iff $\Gamma \cup \{\neg A\}$ is unsatisfiable.*

Proof. For the forward direction, suppose $\Gamma \models A$ and suppose to the contrary that there is a structure M so that $M \models \Gamma \cup \{\neg A\}$. Since $M \models \Gamma$ and $\Gamma \models A$, $M \models A$. Also, since $M \models \Gamma \cup \{\neg A\}$, $M \models \neg A$, so we have both $M \models A$ and $M \not\models A$, a contradiction. Hence, there can be no such structure M , so $\Gamma \cup \{A\}$ is unsatisfiable.

For the reverse direction, suppose $\Gamma \cup \{\neg A\}$ is unsatisfiable. So for every structure M , either $M \not\models \Gamma$ or $M \models A$. Hence, for every structure M with $M \models \Gamma$, $M \models A$, so $\Gamma \models A$. \square

Proposition 5.46. *If $\Gamma \subseteq \Gamma'$ and $\Gamma \models A$, then $\Gamma' \models A$.*

Proof. Suppose that $\Gamma \subseteq \Gamma'$ and $\Gamma \models A$. Let M be such that $M \models \Gamma'$; then $M \models \Gamma$, and since $\Gamma \models A$, we get that $M \models A$. Hence, whenever $M \models \Gamma'$, $M \models A$, so $\Gamma' \models A$. \square

Theorem 5.47 (Semantic Deduction Theorem). *$\Gamma \cup \{A\} \models B$ iff $\Gamma \models A \rightarrow B$.*

Proof. For the forward direction, let $\Gamma \cup \{A\} \models B$ and let M be a structure so that $M \models \Gamma$. If $M \models A$, then $M \models \Gamma \cup \{A\}$, so since $\Gamma \cup \{A\}$ entails B , we get $M \models B$. Therefore, $M \models A \rightarrow B$, so $\Gamma \models A \rightarrow B$.

For the reverse direction, let $\Gamma \models A \rightarrow B$ and M be a structure so that $M \models \Gamma \cup \{A\}$. Then $M \models \Gamma$, so $M \models A \rightarrow B$, and since $M \models A$, $M \models B$. Hence, whenever $M \models \Gamma \cup \{A\}$, $M \models B$, so $\Gamma \cup \{A\} \models B$. \square

CHAPTER 6

Theories and Their Models

6.1 Introduction

The development of the axiomatic method is a significant achievement in the history of science, and is of special importance in the history of mathematics. An axiomatic development of a field involves the clarification of many questions: What is the field about? What are the most fundamental concepts? How are they related? Can all the concepts of the field be defined in terms of these fundamental concepts? What laws do, and must, these concepts obey?

The axiomatic method and logic were made for each other. Formal logic provides the tools for formulating axiomatic theories, for proving theorems from the axioms of the theory in a precisely specified way, for studying the properties of all systems satisfying the axioms in a systematic way.

Definition 6.1. A set of sentences Γ is *closed* iff, whenever $\Gamma \models A$ then $A \in \Gamma$. The *closure* of a set of sentences Γ is $\{A : \Gamma \models A\}$.

We say that Γ is *axiomatized by* a set of sentences Δ if Γ is the closure of Δ .

We can think of an axiomatic theory as the set of sentences that is axiomatized by its set of axioms Δ . In other words, when we have a first-order language which contains non-logical symbols for the primitives of the axiomatically developed science we wish to study, together with a set of sentences that express the fundamental laws of the science, we can think of the theory as represented by all the sentences in this language that are entailed by the axioms. This ranges from simple examples with only a single primitive and simple axioms, such as the theory of partial orders, to complex theories such as Newtonian mechanics.

The important logical facts that make this formal approach to the axiomatic method so important are the following. Suppose Γ is an axiom system for a theory, i.e., a set of sentences.

1. We can state precisely when an axiom system captures an intended class of structures. That is, if we are interested in a certain class of structures, we will successfully capture that class by an axiom system Γ iff the structures are exactly those M such that $M \models \Gamma$.
2. We may fail in this respect because there are M such that $M \models \Gamma$, but M is not one of the structures we intend. This may lead us to add axioms which are not true in M .
3. If we are successful at least in the respect that Γ is true in all the intended structures, then a sentence A is true in all intended structures whenever $\Gamma \models A$. Thus we can use logical tools (such as proof methods) to show that sentences are true in all intended structures simply by showing that they are entailed by the axioms.
4. Sometimes we don't have intended structures in mind, but instead start from the axioms themselves: we begin with some primitives that we want to satisfy certain laws which we codify in an axiom system. One thing that we would like to verify right away is that the axioms do not contradict each other: if they do, there can be no concepts that obey

these laws, and we have tried to set up an incoherent theory. We can verify that this doesn't happen by finding a model of Γ . And if there are models of our theory, we can use logical methods to investigate them, and we can also use logical methods to construct models.

5. The independence of the axioms is likewise an important question. It may happen that one of the axioms is actually a consequence of the others, and so is redundant. We can prove that an axiom A in Γ is redundant by proving $\Gamma \setminus \{A\} \models A$. We can also prove that an axiom is not redundant by showing that $(\Gamma \setminus \{A\}) \cup \{\neg A\}$ is satisfiable. For instance, this is how it was shown that the parallel postulate is independent of the other axioms of geometry.
6. Another important question is that of definability of concepts in a theory: The choice of the language determines what the models of a theory consists of. But not every aspect of a theory must be represented separately in its models. For instance, every ordering \leq determines a corresponding strict ordering $<$ —given one, we can define the other. So it is not necessary that a model of a theory involving such an order must *also* contain the corresponding strict ordering. When is it the case, in general, that one relation can be defined in terms of others? When is it impossible to define a relation in terms of other (and hence must add it to the primitives of the language)?

6.2 Expressing Properties of Structures

It is often useful and important to express conditions on functions and relations, or more generally, that the functions and relations in a structure satisfy these conditions. For instance, we would like to have ways of distinguishing those structures for a language which “capture” what we want the predicate symbols to “mean” from those that do not. Of course we're completely

free to specify which structures we “intend,” e.g., we can specify that the interpretation of the predicate symbol \leq must be an ordering, or that we are only interested in interpretations of \mathcal{L} in which the domain consists of sets and \in is interpreted by the “is an element of” relation. But can we do this with sentences of the language? In other words, which conditions on a structure M can we express by a sentence (or perhaps a set of sentences) in the language of M ? There are some conditions that we will not be able to express. For instance, there is no sentence of \mathcal{L}_A which is only true in a structure M if $|M| = \mathbb{N}$. We cannot express “the domain contains only natural numbers.” But there are “structural properties” of structures that we perhaps can express. Which properties of structures can we express by sentences? Or, to put it another way, which collections of structures can we describe as those making a sentence (or set of sentences) true?

Definition 6.2. Let Γ be a set of sentences in a language \mathcal{L} . We say that a structure M is a *model of Γ* if $M \models A$ for all $A \in \Gamma$.

Example 6.3. The sentence $\forall x x \leq x$ is true in M iff \leq^M is a reflexive relation. The sentence $\forall x \forall y ((x \leq y \wedge y \leq x) \rightarrow x = y)$ is true in M iff \leq^M is anti-symmetric. The sentence $\forall x \forall y \forall z ((x \leq y \wedge y \leq z) \rightarrow x \leq z)$ is true in M iff \leq^M is transitive. Thus, the models of

$$\left\{ \begin{array}{l} \forall x x \leq x, \\ \forall x \forall y ((x \leq y \wedge y \leq x) \rightarrow x = y), \\ \forall x \forall y \forall z ((x \leq y \wedge y \leq z) \rightarrow x \leq z) \end{array} \right\}$$

are exactly those structures in which \leq^M is reflexive, anti-symmetric, and transitive, i.e., a partial order. Hence, we can take them as axioms for the *first-order theory of partial orders*.

6.3 Examples of First-Order Theories

Example 6.4. The theory of strict linear orders in the language $\mathcal{L}_<$ is axiomatized by the set

$$\begin{aligned} &\forall x \neg x < x, \\ &\forall x \forall y ((x < y \vee y < x) \vee x = y), \\ &\forall x \forall y \forall z ((x < y \wedge y < z) \rightarrow x < z) \end{aligned}$$

It completely captures the intended structures: every strict linear order is a model of this axiom system, and vice versa, if R is a linear order on a set X , then the structure M with $|M| = X$ and $<^M = R$ is a model of this theory.

Example 6.5. The theory of groups in the language $\mathbf{1}$ (constant symbol), \cdot (two-place function symbol) is axiomatized by

$$\begin{aligned} &\forall x (x \cdot \mathbf{1}) = x \\ &\forall x \forall y \forall z (x \cdot (y \cdot z)) = ((x \cdot y) \cdot z) \\ &\forall x \exists y (x \cdot y) = \mathbf{1} \end{aligned}$$

Example 6.6. The theory of Peano arithmetic is axiomatized by the following sentences in the language of arithmetic \mathcal{L}_A .

$$\begin{aligned} &\neg \exists x x' = 0 \\ &\forall x \forall y (x' = y' \rightarrow x = y) \\ &\forall x \forall y (x < y \leftrightarrow \exists z (x + z' = y)) \\ &\forall x (x + 0) = x \\ &\forall x \forall y (x + y') = (x + y)' \\ &\forall x (x \times 0) = 0 \\ &\forall x \forall y (x \times y') = ((x \times y) + x) \end{aligned}$$

plus all sentences of the form

$$(A(0) \wedge \forall x (A(x) \rightarrow A(x')))) \rightarrow \forall x A(x)$$

Since there are infinitely many sentences of the latter form, this axiom system is infinite. The latter form is called the *induction schema*. (Actually, the induction schema is a bit more complicated than we let on here.)

The third axiom is an *explicit definition* of $<$.

Example 6.7. The theory of pure sets plays an important role in the foundations (and in the philosophy) of mathematics. A set is pure if all its elements are also pure sets. The empty set counts therefore as pure, but a set that has something as an element that is not a set would not be pure. So the pure sets are those that are formed just from the empty set and no “urelements,” i.e., objects that are not themselves sets.

The following might be considered as an axiom system for a theory of pure sets:

$$\begin{aligned} \exists x \neg \exists y y \in x \\ \forall x \forall y (\forall z (z \in x \leftrightarrow z \in y) \rightarrow x = y) \\ \forall x \forall y \exists z \forall u (u \in z \leftrightarrow (u = x \vee u = y)) \\ \forall x \exists y \forall z (z \in y \leftrightarrow \exists u (z \in u \vee u \in x)) \end{aligned}$$

plus all sentences of the form

$$\exists x \forall y (y \in x \leftrightarrow A(y))$$

The first axiom says that there is a set with no elements (i.e., \emptyset exists); the second says that sets are extensional; the third that for any sets X and Y , the set $\{X, Y\}$ exists; the fourth that for any sets X and Y , the set $X \cup Y$ exists.

The sentences mentioned last are collectively called the *naive comprehension scheme*. It essentially says that for every $A(x)$, the set $\{x : A(x)\}$ exists—so at first glance a true, useful, and perhaps even necessary axiom. It is called “naive” because, as it turns out, it makes this theory unsatisfiable: if you take $A(y)$ to be $\neg y \in y$, you get the sentence

$$\exists x \forall y (y \in x \leftrightarrow \neg y \in y)$$

and this sentence is not satisfied in any structure.

Example 6.8. In the area of *mereology*, the relation of *parthood* is a fundamental relation. Just like theories of sets, there are theories of parthood that axiomatize various conceptions (sometimes conflicting) of this relation.

The language of mereology contains a single two-place predicate symbol P , and $P(x, y)$ “means” that x is a part of y . When we have this interpretation in mind, a structure for this language is called a *parthood structure*. Of course, not every structure for a single two-place predicate will really deserve this name. To have a chance of capturing “parthood,” P^M must satisfy some conditions, which we can lay down as axioms for a theory of parthood. For instance, parthood is a partial order on objects: every object is a part (albeit an *improper* part) of itself; no two different objects can be parts of each other; a part of a part of an object is itself part of that object. Note that in this sense “is a part of” resembles “is a subset of,” but does not resemble “is an element of” which is neither reflexive nor transitive.

$$\begin{aligned} &\forall x P(x, x), \\ &\forall x \forall y ((P(x, y) \wedge P(y, x)) \rightarrow x = y), \\ &\forall x \forall y \forall z ((P(x, y) \wedge P(y, z)) \rightarrow P(x, z)), \end{aligned}$$

Moreover, any two objects have a fusion (an object that has only these two objects and all their parts as parts).

$$\forall x \forall y \exists z \forall u (P(u, z) \leftrightarrow (P(u, x) \wedge P(u, y)))$$

These are only some of the basic principles of parthood considered by metaphysicians. Further principles, however, quickly become hard to formulate or write down without first introducing some defined relations. For instance, most metaphysicians interested in mereology also view the following as a valid principle: whenever an object x has a proper part y , it also has a part z that has no parts in common with y , and so that the fusion of y and z is x .

6.4 Expressing Relations in a Structure

One main use formulas can be put to is to express properties and relations in a structure M in terms of the primitives of the language \mathcal{L} of M . By this we mean the following: the domain of M is a set of objects. The constant symbols, function symbols, and predicate symbols are interpreted in M by some objects in $|M|$, functions on $|M|$, and relations on $|M|$. For instance, if A_0^2 is in \mathcal{L} , then M assigns to it a relation $R = A_0^{2M}$. Then the formula $A_0^2(x_1, x_2)$ *expresses* that very relation, in the following sense: if a variable assignment s maps x_1 to $a \in |M|$ and x_2 to $b \in |M|$, then

$$Rab \quad \text{iff} \quad M, s \models A_0^2(x_1, x_2).$$

Note that we have to involve variable assignments here: we can't just say " Rab iff $M \models A_0^2(a, b)$ " because a and b are not symbols of our language: they are elements of $|M|$.

Since we don't just have atomic formulas, but can combine them using the logical connectives and the quantifiers, more complex formulas can define other relations which aren't directly built into M . We're interested in how to do that, and specifically, which relations we can define in a structure.

Definition 6.9. Let $A(x_1, \dots, x_n)$ be a formula of \mathcal{L} in which only x_1, \dots, x_n occur free, and let M be a structure for \mathcal{L} . $A(x_1, \dots, x_n)$ *expresses the relation* $R \subseteq |M|^n$ iff

$$Ra_1 \dots a_n \quad \text{iff} \quad M, s \models A(x_1, \dots, x_n)$$

for any variable assignment s with $s(x_i) = a_i$ ($i = 1, \dots, n$).

Example 6.10. In the standard model of arithmetic \mathbb{N} , the formula $x_1 < x_2 \vee x_1 = x_2$ expresses the \leq relation on \mathbb{N} . The formula $x_2 = x_1'$ expresses the successor relation, i.e., the relation $R \subseteq \mathbb{N}^2$ where Rnm holds if m is the successor of n . The formula $x_1 = x_2'$ expresses the predecessor relation. The formulas $\exists x_3 (x_3 \neq 0 \wedge x_2 = (x_1 + x_3))$ and $\exists x_3 (x_1 + x_3' = x_2)$ both express the $<$ relation. This means that the predicate symbol $<$ is actually superfluous in the language of arithmetic; it can be defined.

This idea is not just interesting in specific structures, but generally whenever we use a language to describe an intended model or models, i.e., when we consider theories. These theories often only contain a few predicate symbols as basic symbols, but in the domain they are used to describe often many other relations play an important role. If these other relations can be systematically expressed by the relations that interpret the basic predicate symbols of the language, we say we can *define* them in the language.

6.5 The Theory of Sets

Almost all of mathematics can be developed in the theory of sets. Developing mathematics in this theory involves a number of things. First, it requires a set of axioms for the relation \in . A number of different axiom systems have been developed, sometimes with conflicting properties of \in . The axiom system known as **ZFC**, Zermelo-Fraenkel set theory with the axiom of choice stands out: it is by far the most widely used and studied, because it turns out that its axioms suffice to prove almost all the things mathematicians expect to be able to prove. But before that can be established, it first is necessary to make clear how we can even *express* all the things mathematicians would like to express. For starters, the language contains no constant symbols or function symbols, so it seems at first glance unclear that we can talk about particular sets (such as \emptyset or \mathbb{N}), can talk about operations on sets (such as $X \cup Y$ and $\wp(X)$), let alone other constructions which involve things other than sets, such as relations and functions.

To begin with, “is an element of” is not the only relation we are interested in: “is a subset of” seems almost as important. But we can *define* “is a subset of” in terms of “is an element of.” To do this, we have to find a formula $A(x, y)$ in the language of set theory which is satisfied by a pair of sets $\langle X, Y \rangle$ iff $X \subseteq Y$. But X is a subset of Y just in case all elements of X are also elements of Y . So we can define \subseteq by the formula

$$\forall z (z \in x \rightarrow z \in y)$$

Now, whenever we want to use the relation \subseteq in a formula, we could instead use that formula (with x and y suitably replaced, and the bound variable z renamed if necessary). For instance, extensionality of sets means that if any sets x and y are contained in each other, then x and y must be the same set. This can be expressed by $\forall x \forall y ((x \subseteq y \wedge y \subseteq x) \rightarrow x = y)$, or, if we replace \subseteq by the above definition, by

$$\forall x \forall y ((\forall z (z \in x \rightarrow z \in y) \wedge \forall z (z \in y \rightarrow z \in x)) \rightarrow x = y).$$

This is in fact one of the axioms of **ZFC**, the “axiom of extensionality.”

There is no constant symbol for \emptyset , but we can express “ x is empty” by $\neg \exists y y \in x$. Then “ \emptyset exists” becomes the sentence $\exists x \neg \exists y y \in x$. This is another axiom of **ZFC**. (Note that the axiom of extensionality implies that there is only one empty set.) Whenever we want to talk about \emptyset in the language of set theory, we would write this as “there is a set that’s empty and . . .” As an example, to express the fact that \emptyset is a subset of every set, we could write

$$\exists x (\neg \exists y y \in x \wedge \forall z x \subseteq z)$$

where, of course, $x \subseteq z$ would in turn have to be replaced by its definition.

To talk about operations on sets, such as $X \cup Y$ and $\wp(X)$, we have to use a similar trick. There are no function symbols in the language of set theory, but we can express the functional relations $X \cup Y = Z$ and $\wp(X) = Y$ by

$$\begin{aligned} \forall u ((u \in x \vee u \in y) &\leftrightarrow u \in z) \\ \forall u (u \subseteq x &\leftrightarrow u \in y) \end{aligned}$$

since the elements of $X \cup Y$ are exactly the sets that are either elements of X or elements of Y , and the elements of $\wp(X)$ are exactly the subsets of X . However, this doesn’t allow us to use $x \cup y$ or $\wp(x)$ as if they were terms: we can only use the entire formulas that define the relations $X \cup Y = Z$ and $\wp(X) = Y$. In

fact, we do not know that these relations are ever satisfied, i.e., we do not know that unions and power sets always exist. For instance, the sentence $\forall x \exists y \wp(x) = y$ is another axiom of **ZFC** (the power set axiom).

Now what about talk of ordered pairs or functions? Here we have to explain how we can think of ordered pairs and functions as special kinds of sets. One way to define the ordered pair $\langle x, y \rangle$ is as the set $\{\{x\}, \{x, y\}\}$. But like before, we cannot introduce a function symbol that names this set; we can only define the relation $\langle x, y \rangle = z$, i.e., $\{\{x\}, \{x, y\}\} = z$:

$$\forall u (u \in z \leftrightarrow (\forall v (v \in u \leftrightarrow v = x) \vee \forall v (v \in u \leftrightarrow (v = x \vee v = y))))$$

This says that the elements u of z are exactly those sets which either have x as its only element or have x and y as its only elements (in other words, those sets that are either identical to $\{x\}$ or identical to $\{x, y\}$). Once we have this, we can say further things, e.g., that $X \times Y = Z$:

$$\forall z (z \in Z \leftrightarrow \exists x \exists y (x \in X \wedge y \in Y \wedge \langle x, y \rangle = z))$$

A function $f: X \rightarrow Y$ can be thought of as the relation $f(x) = y$, i.e., as the set of pairs $\{\langle x, y \rangle : f(x) = y\}$. We can then say that a set f is a function from X to Y if (a) it is a relation $\subseteq X \times Y$, (b) it is total, i.e., for all $x \in X$ there is some $y \in Y$ such that $\langle x, y \rangle \in f$ and (c) it is functional, i.e., whenever $\langle x, y \rangle, \langle x, y' \rangle \in f$, $y = y'$ (because values of functions must be unique). So “ f is a function from X to Y ” can be written as:

$$\begin{aligned} &\forall u (u \in f \rightarrow \exists x \exists y (x \in X \wedge y \in Y \wedge \langle x, y \rangle = u)) \wedge \\ &\forall x (x \in X \rightarrow (\exists y (y \in Y \wedge \text{maps}(f, x, y)) \wedge \\ &\quad (\forall y \forall y' ((\text{maps}(f, x, y) \wedge \text{maps}(f, x, y')) \rightarrow y = y')))) \end{aligned}$$

where $\text{maps}(f, x, y)$ abbreviates $\exists v (v \in f \wedge \langle x, y \rangle = v)$ (this formula expresses “ $f(x) = y$ ”).

It is now also not hard to express that $f: X \rightarrow Y$ is injective, for instance:

$$f: X \rightarrow Y \wedge \forall x \forall x' ((x \in X \wedge x' \in X \wedge \exists y (\text{maps}(f, x, y) \wedge \text{maps}(f, x', y))) \rightarrow x = x')$$

A function $f: X \rightarrow Y$ is injective iff, whenever f maps $x, x' \in X$ to a single y , $x = x'$. If we abbreviate this formula as $\text{inj}(f, X, Y)$, we're already in a position to state in the language of set theory something as non-trivial as Cantor's theorem: there is no injective function from $\wp(X)$ to X :

$$\forall X \forall Y (\wp(X) = Y \rightarrow \neg \exists f \text{inj}(f, Y, X))$$

6.6 Expressing the Size of Structures

There are some properties of structures we can express even without using the non-logical symbols of a language. For instance, there are sentences which are true in a structure iff the domain of the structure has at least, at most, or exactly a certain number n of elements.

Proposition 6.11. *The sentence*

$$\begin{aligned} A_{\geq n} \equiv \exists x_1 \exists x_2 \dots \exists x_n \quad & (x_1 \neq x_2 \wedge x_1 \neq x_3 \wedge x_1 \neq x_4 \wedge \dots \wedge x_1 \neq x_n \wedge \\ & x_2 \neq x_3 \wedge x_2 \neq x_4 \wedge \dots \wedge x_2 \neq x_n \wedge \\ & \vdots \\ & x_{n-1} \neq x_n) \end{aligned}$$

is true in a structure M iff $|M|$ contains at least n elements. Consequently, $M \models \neg A_{\geq n+1}$ iff $|M|$ contains at most n elements.

Proposition 6.12. *The sentence*

$$\begin{aligned}
 A_{=n} \equiv \exists x_1 \exists x_2 \dots \exists x_n \quad & (x_1 \neq x_2 \wedge x_1 \neq x_3 \wedge x_1 \neq x_4 \wedge \dots \wedge x_1 \neq x_n \wedge \\
 & x_2 \neq x_3 \wedge x_2 \neq x_4 \wedge \dots \wedge x_2 \neq x_n \wedge \\
 & \vdots \\
 & x_{n-1} \neq x_n \wedge \\
 & \forall y (y = x_1 \vee \dots \vee y = x_n) \dots)
 \end{aligned}$$

is true in a structure M iff $|M|$ contains exactly n elements.

Proposition 6.13. *A structure is infinite iff it is a model of*

$$\{A_{\geq 1}, A_{\geq 2}, A_{\geq 3}, \dots\}$$

There is no single purely logical sentence which is true in M iff $|M|$ is infinite. However, one can give sentences with non-logical predicate symbols which only have infinite models (although not every infinite structure is a model of them). The property of being a finite structure, and the property of being a uncountable structure cannot even be expressed with an infinite set of sentences. These facts follow from the compactness and Löwenheim-Skolem theorems.

CHAPTER 7

Natural Deduction

7.1 Rules and Derivations

Let \mathcal{L} be a first-order language with the usual constants, variables, logical symbols, and auxiliary symbols (parentheses and the comma).

Definition 7.1 (Inference). An *inference* is an expression of the form

$$\frac{A}{C} \quad \text{or} \quad \frac{A \quad B}{C}$$

where A, B , and C are formulas. A and B are called the *upper formulas* or *premises* and C the *lower formulas* or *conclusion* of the inference.

The rules for natural deduction are divided into two main types: *propositional* rules (quantifier-free) and *quantifier* rules. The rules come in pairs, an introduction and an elimination rule for each logical operator. They introduced an logical operator in the conclusion or remove and logical operator from a premise of the rule. Some of the rules allow an assumption of a certain type to be *discharged*. To indicate which assumption is discharged

by which inference, we also assign labels to both the assumption and the inference. This is indicated by writing the assumption formula as “[A]^{*n*}”.

It is customary to consider rules for all logical operators, even for those (if any) that we consider as defined.

Propositional Rules

Rules for \perp

$$\frac{A \quad \neg A}{\perp} \perp \text{Intro} \quad \frac{\perp}{A} \perp \text{Elim}$$

Rules for \wedge

$$\frac{A \quad B}{A \wedge B} \wedge \text{Intro} \quad \frac{A \wedge B}{A} \wedge \text{Elim} \quad \frac{A \wedge B}{B} \wedge \text{Elim}$$

Rules for \vee

$$\frac{A}{A \vee B} \vee \text{Intro} \quad \frac{B}{A \vee B} \vee \text{Intro} \quad \begin{array}{c} [A]^n \quad [B]^n \\ \vdots \quad \vdots \\ n \quad \frac{A \vee B \quad C \quad C}{C} \vee \text{Elim} \end{array}$$

Rules for \neg

$$\begin{array}{c} [A]^n \\ \vdots \\ n \quad \frac{\perp}{\neg A} \neg \text{Intro} \end{array} \quad \frac{\neg \neg A}{A} \neg \text{Elim}$$

Rules for \rightarrow

$$\begin{array}{c} [A]^n \\ \vdots \\ n \quad \frac{B}{A \rightarrow B} \rightarrow \text{Intro} \end{array} \quad \frac{A \quad A \rightarrow B}{B} \rightarrow \text{Elim}$$

Quantifier Rules

Rules for \forall

$$\frac{A(a)}{\forall x A(x)} \forall\text{Intro} \quad \frac{\forall x A(x)}{A(t)} \forall\text{Elim}$$

where t is a ground term, and a is a constant which does not occur in A , or in any assumption which is undischarged in the derivation ending with the premise A . We call a the *eigenvariable* of the $\forall\text{Intro}$ inference.

Rules for \exists

$$\frac{A(a)}{\exists x A(x)} \exists\text{Intro} \quad \frac{\exists x A(x) \quad \begin{array}{c} [A(a)]^n \\ \vdots \\ C \end{array}}{C} \exists\text{Elim}_n$$

where t is a ground term, and a is a constant which does not occur in the premise $\exists x A(x)$, in C , or any assumption which is undischarged in the derivations ending with the two premises C (other than the assumptions $A(a)$). We call a the *eigenvariable* of the $\exists\text{Elim}$ inference.

The condition that an eigenvariable not occur in the upper sequent of the \forall intro or \exists elim inference is called the *eigenvariable condition*.

We use the term “eigenvariable” even though a in the above rules is a constant. This has historical reasons.

In $\exists\text{Intro}$ and $\forall\text{Elim}$ there are no restrictions, and the term t can be anything, so we do not have to worry about any conditions. However, because the t may appear elsewhere in the derivation, the values of t for which the formula is satisfied are constrained. On the other hand, in the $\exists\text{Elim}$ and \forall intro rules, the eigenvariable condition requires that a does not occur anywhere else in the formula. Thus, if the upper formula is valid, the truth values of the formulas other than $A(a)$ are independent of a .

Natural deduction systems are meant to closely parallel the informal reasoning used in mathematical proof (hence it is somewhat “natural”). Natural deduction proofs begin with assumptions. Inference rules are then applied. Assumptions are “discharged” by the \neg Intro, \rightarrow Intro, \vee Elim and \exists Elim inference rules, and the label of the discharged assumption is placed beside the inference for clarity.

Definition 7.2 (Initial Formula). An *initial formula* or *assumption* is any formula in the topmost position of any branch.

Definition 7.3 (Derivation). A *derivation* of a formula A from assumptions Γ is a tree of formulas satisfying the following conditions:

1. The topmost formulas of the tree are either in Γ or are discharged by an inference in the tree.
2. Every formula in the tree is an upper formula of an inference whose lower formula stands directly below that formula in the tree.

We then say that A is the *end-formula* of the derivation and that A is *derivable* from Γ .

Definition 7.4 (Theorem). A sentence A is a *theorem* if it is derivable from the empty set.

7.2 Examples of Derivations

Example 7.5. Let’s give a derivation of the formula $(A \wedge B) \rightarrow A$.

We begin by writing the desired end-formula at the bottom of the derivation.

$$\overline{(A \wedge B) \rightarrow A}$$

Next, we need to figure out what kind of inference could result in a formula of this form. The main operator of the end-formula is \rightarrow , so we’ll try to arrive at the end-formula using the \rightarrow Intro

rule. It is best to write down the assumptions involved and label the inference rules as you progress, so it is easy to see whether all assumptions have been discharged at the end of the proof.

$$\begin{array}{c} [A \wedge B]^1 \\ \vdots \\ A \\ 1 \frac{}{(A \wedge B) \rightarrow A} \rightarrow \text{Intro} \end{array}$$

We now need to fill in the steps from the assumption $A \wedge B$ to A . Since we only have one connective to deal with, \wedge , we must use the \wedge elim rule. This gives us the following proof:

$$\begin{array}{c} [A \wedge B]^1 \\ A \wedge \text{Elim} \\ 1 \frac{}{(A \wedge B) \rightarrow A} \rightarrow \text{Intro} \end{array}$$

We now have a correct derivation of the formula $(A \wedge B) \rightarrow A$.

Example 7.6. Now let's give a derivation of the formula $(\neg A \vee B) \rightarrow (A \rightarrow B)$.

We begin by writing the desired end-formula at the bottom of the derivation.

$$\overline{(\neg A \vee B) \rightarrow (A \rightarrow B)}$$

To find a logical rule that could give us this end-formula, we look at the logical connectives in the end-formula: \neg , \vee , and \rightarrow . We only care at the moment about the first occurrence of \rightarrow because it is the main operator of the sentence in the end-sequent, while \neg , \vee and the second occurrence of \rightarrow are inside the scope of another connective, so we will take care of those later. We therefore start with the \rightarrow Intro rule. A correct application must look as follows:

$$\begin{array}{c} [\neg A \vee B]^1 \\ \vdots \\ A \rightarrow B \\ 1 \frac{}{(\neg A \vee B) \rightarrow (A \rightarrow B)} \rightarrow \text{Intro} \end{array}$$

This leaves us with two possibilities to continue. Either we can keep working from the bottom up and look for another application of the \rightarrow Intro rule, or we can work from the top down and apply a \vee Elim rule. Let us apply the latter. We will use the assumption $\neg A \vee B$ as the leftmost premise of \vee Elim. For a valid application of \vee Elim, the other two premises must be identical to the conclusion $A \rightarrow B$, but each may be derived in turn from another assumption, namely the two disjuncts of $\neg A \vee B$. So our derivation will look like this:

$$\begin{array}{c}
 \begin{array}{cc}
 [\neg A]^2 & [B]^2 \\
 \vdots & \vdots \\
 \frac{[\neg A \vee B]^1 \quad A \rightarrow B \quad A \rightarrow B}{A \rightarrow B} \vee\text{Elim} \\
 \frac{A \rightarrow B}{(\neg A \vee B) \rightarrow (A \rightarrow B)} \rightarrow \text{Intro}
 \end{array}
 \end{array}$$

In each of the two branches on the right, we want to derive $A \rightarrow B$, which is best done using \rightarrow Intro.

$$\begin{array}{c}
 \begin{array}{ccc}
 [\neg A]^2, [A]^3 & & [B]^2, [A]^4 \\
 \vdots & & \vdots \\
 \frac{B}{A \rightarrow B} \rightarrow \text{Intro} & & \frac{B}{A \rightarrow B} \rightarrow \text{Intro} \\
 \frac{[\neg A \vee B]^1 \quad A \rightarrow B \quad A \rightarrow B}{A \rightarrow B} \vee\text{Elim} \\
 \frac{A \rightarrow B}{(\neg A \vee B) \rightarrow (A \rightarrow B)} \rightarrow \text{Intro}
 \end{array}
 \end{array}$$

For the two missing parts of the derivation, we need derivations of B from $\neg A$ and A in the middle, and from A and B on the left. Let's take the former first. $\neg A$ and A are the two premises of \perp Intro:

$$\begin{array}{c}
 \frac{[\neg A]^2 \quad [A]^3}{\perp} \perp \text{Intro} \\
 \vdots \\
 B
 \end{array}$$

By using \perp Elim, we can obtain B as a conclusion and complete the branch.

$$\begin{array}{c}
 \begin{array}{c}
 \frac{[\neg A]^2 \quad [A]^3}{\perp} \perp \text{Intro} \\
 \frac{\perp}{B} \perp \text{Elim} \\
 \frac{[\neg A \vee B]^1 \quad \frac{A \rightarrow B}{3} \rightarrow \text{Intro} \quad \frac{B}{4} \rightarrow \text{Intro}}{A \rightarrow B} \vee \text{Elim} \\
 \frac{A \rightarrow B}{1} (\neg A \vee B) \rightarrow (A \rightarrow B) \rightarrow \text{Intro}
 \end{array}
 \end{array}$$

$[B]^2, [A]^4$
 \vdots

Let's now look at the rightmost branch. Here it's important to realize that the definition of derivation *allows assumptions to be discharged* but *does not require* them to be. In other words, if we can derive B from one of the assumptions A and B without using the other, that's ok. And to derive B from B is trivial: B by itself is such a derivation, and no inferences are needed. So we can simply delete the assumption A .

$$\begin{array}{c}
 \begin{array}{c}
 \frac{[\neg A]^2 \quad [A]^3}{\perp} \perp \text{Intro} \\
 \frac{\perp}{B} \perp \text{Elim} \\
 \frac{[\neg A \vee B]^1 \quad \frac{A \rightarrow B}{3} \rightarrow \text{Intro} \quad \frac{[B]^2}{A \rightarrow B} \rightarrow \text{Intro}}{A \rightarrow B} \vee \text{Elim} \\
 \frac{A \rightarrow B}{1} (\neg A \vee B) \rightarrow (A \rightarrow B) \rightarrow \text{Intro}
 \end{array}
 \end{array}$$

Note that in the finished derivation, the rightmost \rightarrow Intro inference does not actually discharge any assumptions.

Example 7.7. When dealing with quantifiers, we have to make sure not to violate the eigenvariable condition, and sometimes this requires us to play around with the order of carrying out certain inferences. In general, it helps to try and take care of rules subject to the eigenvariable condition first (they will be lower down in the finished proof).

Let's see how we'd give a derivation of the formula $\exists x \neg A(x) \rightarrow \neg \forall x A(x)$. Starting as usual, we write

$$\overline{\exists x \neg A(x) \rightarrow \neg \forall x A(x)}$$

We start by writing down what it would take to justify that last step using the \rightarrow Intro rule.

$$\frac{\begin{array}{c} [\exists x \neg A(x)]^1 \\ \vdots \\ \neg \forall x A(x) \end{array}}{\exists x \neg A(x) \rightarrow \neg \forall x A(x)} \rightarrow \text{Intro}$$

Since there is no obvious rule to apply to $\neg \forall x A(x)$, we will proceed by setting up the derivation so we can use the \exists Elim rule. Here we must pay attention to the eigenvariable condition, and choose a constant that does not appear in $\exists x A(x)$ or any assumptions that it depends on. (Since no constant symbols appear, however, any choice will do fine.)

$$\begin{array}{c} [\neg A(a)]^2 \\ \vdots \\ \frac{2 \frac{[\exists x \neg A(x)]^1 \quad \neg \forall x A(x)}{\neg \forall x A(x)} \exists \text{Elim}}{\exists x \neg A(x) \rightarrow \neg \forall x A(x)} \rightarrow \text{Intro} \end{array}$$

In order to derive $\neg \forall x A(x)$, we will attempt to use the \neg Intro rule: this requires that we derive a contradiction, possibly using $\forall x A(x)$ as an additional assumption. Of course, this contradiction may involve the assumption $\neg A(a)$ which will be discharged by the \rightarrow Intro inference. We can set it up as follows:

$$\begin{array}{c} [\neg A(a)]^2, [\forall x A(x)]^3 \\ \vdots \\ \frac{2 \frac{[\exists x \neg A(x)]^1 \quad 3 \frac{\perp}{\neg \forall x A(x)} \neg \text{Intro}}{\neg \forall x A(x)} \exists \text{Elim}}{\exists x \neg A(x) \rightarrow \neg \forall x A(x)} \rightarrow \text{Intro} \end{array}$$

It looks like we are close to getting a contradiction. The easiest rule to apply is the \forall Elim, which has no eigenvariable conditions.

Since we can use any term we want to replace the universally quantified x , it makes the most sense to continue using a so we can reach a contradiction.

$$\begin{array}{c}
 \frac{\frac{\frac{[\neg A(a)]^2}{\frac{[\forall x A(x)]^3}{A(a)} \text{ } \forall\text{Elim}}{\perp} \text{ } \perp\text{Intro}}{\neg\forall x A(x)} \text{ } \neg\text{Intro}}{\neg\forall x A(x)} \text{ } \exists\text{Elim} \\
 \frac{[\exists x \neg A(x)]^1}{\exists x \neg A(x) \rightarrow \neg\forall x A(x)} \text{ } \rightarrow\text{Intro}
 \end{array}$$

It is important, especially when dealing with quantifiers, to double check at this point that the eigenvariable condition has not been violated. Since the only rule we applied that is subject to the eigenvariable condition was $\exists\text{Elim}$, and the eigenvariable a does not occur in any assumptions it depends on, this is a correct derivation.

7.3 Proof-Theoretic Notions

Just as we've defined a number of important semantic notions (validity, entailment, satisfiability), we now define corresponding *proof-theoretic notions*. These are not defined by appeal to satisfaction of sentences in structures, but by appeal to the derivability or non-derivability of certain formulas. It was an important discovery, due to Gödel, that these notions coincide. That they do is the content of the *completeness theorem*.

Definition 7.8 (Derivability). A formula A is *derivable* from a set of formulas Γ , $\Gamma \vdash A$, if there is a derivation with end-formula A and in which every assumption is either discharged or is in Γ . If A is not derivable from Γ we write $\Gamma \not\vdash A$.

Definition 7.9 (Theorems). A formula A is a *theorem* if there is a derivation of A from the empty set. We write $\vdash A$ if A is a theorem and $\not\vdash A$ if it is not.

Definition 7.10 (Consistency). A set of sentences Γ is *consistent* iff $\Gamma \not\vdash \perp$. If Γ is not consistent, i.e., if $\Gamma \vdash \perp$, we say it is *inconsistent*.

Proposition 7.11. $\Gamma \vdash A$ iff $\Gamma \cup \{\neg A\}$ is inconsistent.

Proof. Exercise. □

Proposition 7.12. Γ is inconsistent iff $\Gamma \vdash A$ for every sentence A .

Proof. Exercise. □

Proposition 7.13. If $\Gamma \vdash A$ iff for some finite $\Gamma_0 \subseteq \Gamma$, $\Gamma_0 \vdash A$.

Proof. Any derivation of A from Γ can only contain finitely many undischarged assumptions. If all these undischarged assumptions are in Γ , then the set of them is a finite subset of Γ . The other direction is trivial, since a derivation from a subset of Γ is also a derivation from Γ . □

7.4 Properties of Derivability

We will now establish a number of properties of the derivability relation. They are independently interesting, but each will play a role in the proof of the completeness theorem.

Proposition 7.14 (Monotony). If $\Gamma \subseteq \Delta$ and $\Gamma \vdash A$, then $\Delta \vdash A$.

Proof. Any derivation of A from Γ is also a derivation of A from Δ . □

Proposition 7.15. If $\Gamma \vdash A$ and $\Gamma \cup \{A\} \vdash \perp$, then Γ is inconsistent.

$$\begin{array}{c}
 [A]^1 \\
 \vdots \delta_2 \\
 \vdots \delta_1 \\
 A \quad \quad \quad \perp \\
 \hline
 \perp
 \end{array}
 \begin{array}{l}
 \frac{}{\neg A} \text{Intro} \\
 \frac{}{\neg A} \text{Elim}
 \end{array}$$

Proposition 7.16. *If $\Gamma \cup \{A\} \vdash \perp$, then $\Gamma \vdash \neg A$.*

$$\begin{array}{c} [A]^1 \\ \vdots \\ \vdots \delta \\ \vdots \\ \perp \\ 1 \frac{}{\neg A} \neg \text{Intro} \end{array}$$
$$\frac{\begin{array}{c} [A]^1 \\ \vdots \\ \delta_1 \\ \perp \end{array} \quad \begin{array}{c} [\neg A]^2 \\ \vdots \\ \delta_2 \\ \perp \end{array} \quad \begin{array}{c} 1 \\ \neg A \end{array} \quad \neg\text{Intro} \quad \frac{\quad}{\perp} \quad \begin{array}{c} 2 \\ \neg A \end{array} \quad \neg\text{Intro} \quad \neg\text{Elim}$$

Since the assumptions A and $\neg A$ are discharged, this is a derivation from Γ alone. Hence $\Gamma \vdash \perp$. \square

Proposition 7.18. *If $\Gamma \cup \{A\} \vdash \perp$ and $\Gamma \cup \{B\} \vdash \perp$, then $\Gamma \cup \{A \vee B\} \vdash \perp$.*

Proof. Exercise. □

Proposition 7.19. *If $\Gamma \vdash A$ or $\Gamma \vdash B$, then $\Gamma \vdash A \vee B$.*

Proof. Suppose $\Gamma \vdash A$. There is a derivation δ of A from Γ . We can derive

$$\frac{\begin{array}{c} \vdots \\ \vdots \delta \\ \vdots \\ A \end{array}}{A \vee B} \vee\text{Intro}$$

Therefore $\Gamma \vdash A \vee B$. The proof for when $\Gamma \vdash B$ is similar. □

Proposition 7.20. *If $\Gamma \vdash A \wedge B$ then $\Gamma \vdash A$ and $\Gamma \vdash B$.*

Proof. Exercise □

Proposition 7.21. *If $\Gamma \vdash A$ and $\Gamma \vdash B$, then $\Gamma \vdash A \wedge B$.*

Proof. Exercise. □

Proposition 7.22. *If $\Gamma \vdash A$ and $\Gamma \vdash A \rightarrow B$, then $\Gamma \vdash B$.*

Proof. Exercise. □

Proposition 7.23. *If $\Gamma \vdash \neg A$ or $\Gamma \vdash B$, then $\Gamma \vdash A \rightarrow B$.*

Proof. Exercise. □

Theorem 7.24. *If c is a constant not occurring in Γ or $A(x)$ and $\Gamma \vdash A(c)$, then $\Gamma \vdash \forall x A(x)$.*

Proof. Let δ be an derivation of $A(c)$ from Γ . By adding a \forall Intro inference, we obtain a proof of $\forall x A(x)$. Since c does not occur in Γ or $A(x)$, the eigenvariable condition is satisfied. \square

Theorem 7.25. 1. *If $\Gamma \vdash A(t)$ then $\Gamma \vdash \exists x A(x)$.*
 2. *If $\Gamma \vdash \forall x A(x)$ then $\Gamma \vdash A(t)$.*

Proof. 1. Suppose $\Gamma \vdash A(t)$. Then there is a derivation δ of $A(t)$ from Γ . The derivation

$$\frac{\begin{array}{c} \vdots \\ \vdots \delta \\ \vdots \\ A(t) \end{array}}{\exists x A(x)} \exists\text{Intro}$$

shows that $\Gamma \vdash \exists x A(x)$.

2. Suppose $\Gamma \vdash \forall x A(x)$. Then there is a derivation δ of $\forall x A(x)$ from Γ . The derivation

$$\frac{\begin{array}{c} \vdots \\ \vdots \delta \\ \vdots \\ \forall x A(x) \end{array}}{A(t)} \forall\text{Elim}$$

shows that $\Gamma \vdash A(t)$.

\square

7.5 Soundness

A derivation system, such as natural deduction, is *sound* if it cannot derive things that do not actually follow. Soundness is

thus a kind of guaranteed safety property for derivation systems. Depending on which proof theoretic property is in question, we would like to know for instance, that

1. every derivable sentence is valid;
2. if a sentence is derivable from some others, it is also a consequence of them;
3. if a set of sentences is inconsistent, it is unsatisfiable.

These are important properties of a derivation system. If any of them do not hold, the derivation system is deficient—it would derive too much. Consequently, establishing the soundness of a derivation system is of the utmost importance.

Theorem 7.26 (Soundness). *If A is derivable from the undischarged assumptions Γ , then $\Gamma \models A$.*

Proof. Inductive Hypothesis: The premises of an inference rule follow from the undischarged assumptions of the subproofs ending in those premises.

Inductive Step: Show that A follows from the undischarged assumptions of the entire proof.

Let δ be a derivation of A . We proceed by induction on the number of inferences in δ .

If the number of inferences is 0, then δ consists only of an initial formula. Every initial formula A is an undischarged assumption, and as such, any structure M that satisfies all of the undischarged assumptions of the proof also satisfies A .

If the number of inferences is greater than 0, we distinguish cases according to the type of the lowermost inference. By induction hypothesis, we can assume that the premises of that inference follow from the undischarged assumptions of the sub-derivations ending in those premises, respectively.

First, we consider the possible inferences with only one premise.

1. Suppose that the last inference is \neg Intro: By inductive hypothesis, \perp follows from the undischarged assumptions $\Gamma \cup \{A\}$. Consider a structure M . We need to show that, if $M \models \Gamma$, then $M \models \neg A$. Suppose for reductio that $M \models \Gamma$, but $M \not\models \neg A$, i.e., $M \models A$. This would mean that $M \models \Gamma \cup \{A\}$. This is contrary to our inductive hypothesis. So, $M \models \neg A$.
2. The last inference is \neg Elim: Exercise.
3. The last inference is \wedge Elim: There are two variants: A or B may be inferred from the premise $A \wedge B$. Consider the first case. By inductive hypothesis, $A \wedge B$ follows from the undischarged assumptions Γ . Consider a structure M . We need to show that, if $M \models \Gamma$, then $M \models A$. By our inductive hypothesis, we know that $M \models A \wedge B$. So, $M \models A$. The case where B is inferred from $A \wedge B$ is handled similarly.
4. The last inference is \vee Intro: There are two variants: $A \vee B$ may be inferred from the premise A or the premise B . Consider the first case. By inductive hypothesis, A follows from the undischarged assumptions Γ . Consider a structure M . We need to show that, if $M \models \Gamma$, then $M \models A \vee B$. Since $M \models \Gamma$, it must be the case that $M \models A$, by inductive hypothesis. So it must also be the case that $M \models A \vee B$. The case where $A \vee B$ is inferred from B is handled similarly.
5. The last inference is \rightarrow Intro: $A \rightarrow B$ is inferred from a sub-proof with assumption A and conclusion B . By inductive hypothesis, B follows from the undischarged assumptions Γ and A . Consider a structure M . We need to show that, if $\Gamma \models A \rightarrow B$. For reductio, suppose that for some structure M , $M \models \Gamma$ but $M \not\models A \rightarrow B$. So, $M \models A$ and $M \not\models B$. But by hypothesis, B is a consequence of $\Gamma \cup \{A\}$. So, $M \models A \rightarrow B$.
6. The last inference is \forall Intro: The premise $A(a)$ is a consequence of the undischarged assumptions Γ by induction

hypothesis. Consider some structure, M , such that $M \models \Gamma$. Let M' be exactly like M except that $a^M \neq a^{M'}$. We must have $M' \models A(a)$.

We now show that $M \models \forall x A(x)$. Since $\forall x A(x)$ is a sentence, this means we have to show that for every variable assignment s , $M, s \models A(x)$. Since Γ consists entirely of sentences, $M, s \models B$ for all $B \in \Gamma$. Let M' be like M except that $a^{M'} = s(x)$. Then $M, s \models A(x)$ iff $M' \models A(a)$ (as $A(x)$ does not contain a). Since a also does not occur in Γ , $M' \models \Gamma$. Since $\Gamma \models A(a)$, $M' \models A(a)$. This means that $M, s \models A(x)$. Since s is an arbitrary variable assignment, $M \models \forall x A(x)$.

7. The last inference is \exists Intro: Exercise.
8. The last inference is \forall Elim: Exercise.

Now let's consider the possible inferences with several premises: \forall Elim, \wedge Intro, \rightarrow Elim, and \exists Elim.

1. The last inference is \wedge Intro. $A \wedge B$ is inferred from the premises A and B . By induction hypothesis, A follows from the undischarged assumptions Γ and B follows from the undischarged assumptions Δ . We have to show that $\Gamma \cup \Delta \models A \wedge B$. Consider a structure M with $M \models \Gamma \cup \Delta$. Since $M \models \Gamma$, it must be the case that $M \models A$, and since $M \models \Delta$, $M \models B$, by inductive hypothesis. Together, $M \models A \wedge B$.
2. The last inference is \forall Elim: Exercise.
3. The last inference is \rightarrow Elim. B is inferred from the premises $A \rightarrow B$ and A . By induction hypothesis, $A \rightarrow B$ follows from the undischarged assumptions Γ and A follows from the undischarged assumptions Δ . Consider a structure M . We need to show that, if $M \models \Gamma \cup \Delta$, then $M \models B$. It must be the case that $M \models A \rightarrow B$, and $M \models A$, by inductive hypothesis. Thus it must be the case that $M \models B$.
4. The last inference is \exists Elim: Exercise.

□

Corollary 7.27. *If $\vdash A$, then A is valid.*

Corollary 7.28. *If Γ is satisfiable, then it is consistent.*

Proof. We prove the contrapositive. Suppose that Γ is not consistent. Then $\Gamma \vdash \perp$, i.e., there is a derivation of \perp from undischarged assumptions in Γ . By [Theorem 7.26](#), any structure M that satisfies Γ must satisfy \perp . Since $M \not\models \perp$ for every structure M , no M can satisfy Γ , i.e., Γ is not satisfiable. □

7.6 Derivations with Identity predicate

Derivations with the identity predicate require additional inference rules.

Rules for $=$:

$$\frac{}{t = t} = \text{Intro}$$

$$\frac{t_1 = t_2 \quad A(t_1)}{A(t_2)} = \text{Elim} \quad \text{and} \quad \frac{t_1 = t_2 \quad A(t_2)}{A(t_1)} = \text{Elim}$$

where t_1 and t_2 are closed terms. The $=$ Intro rule allows us to derive any identity statement of the form $t = t$ outright.

Example 7.29. If s and t are closed terms, then $A(s), s = t \vdash A(t)$:

$$\frac{A(s) \quad s = t}{A(t)} = \text{Elim}$$

This may be familiar as the “principle of substitutability of identicals,” or Leibniz’ Law.

Proposition 7.30. *Natural deduction with rules for identity is sound.*

Proof. Any formula of the form $t = t$ is valid, since for every structure \mathbf{M} , $\mathbf{M} \models t = t$. (Note that we assume the term t to be ground, i.e., it contains no variables, so variable assignments are irrelevant).

Suppose the last inference in a derivation is $=$ Elim. Then the premises are $t_1 = t_2$ and $A(t_1)$; they are derived from undischarged assumptions Γ and Δ , respectively. We want to show that $A(s)$ follows from $\Gamma \cup \Delta$. Consider a structure \mathbf{M} with $\mathbf{M} \models \Gamma \cup \Delta$. By induction hypothesis, \mathbf{M} satisfies the two premises by induction hypothesis. So, $\mathbf{M} \models t_1 = t_2$. Therefore, $\text{Val}^{\mathbf{M}}(t_1) = \text{Val}^{\mathbf{M}}(t_2)$. Let s be any variable assignment, and s' be the x -variant given by $s'(x) = \text{Val}^{\mathbf{M}}(t_1) = \text{Val}^{\mathbf{M}}(t_2)$. By [Proposition 5.40](#), $\mathbf{M}, s \models A(t_2)$ iff $\mathbf{M}, s' \models A(x)$ iff $\mathbf{M}, s \models A(t_1)$. Since $\mathbf{M} \models A(t_1)$ therefore $\mathbf{M} \models A(t_2)$. \square

CHAPTER 8

The Completeness Theorem

8.1 Introduction

The completeness theorem is one of the most fundamental results about logic. It comes in two formulations, the equivalence of which we'll prove. In its first formulation it says something fundamental about the relationship between semantic consequence and our proof system: if a sentence A follows from some sentences Γ , then there is also a derivation that establishes $\Gamma \vdash A$. Thus, the proof system is as strong as it can possibly be without proving things that don't actually follow. In its second formulation, it can be stated as a model existence result: every consistent set of sentences is satisfiable.

These aren't the only reasons the completeness theorem—or rather, its proof—is important. It has a number of important consequences, some of which we'll discuss separately. For instance, since any derivation that shows $\Gamma \vdash A$ is finite and so can only

use finitely many of the sentences in Γ , it follows by the completeness theorem that if A is a consequence of Γ , it is already a consequence of a finite subset of Γ . This is called *compactness*. Equivalently, if every finite subset of Γ is consistent, then Γ itself must be consistent. It also follows from *the proof of* the completeness theorem that any satisfiable set of sentences has a finite or countably infinite model. This result is called the Löwenheim-Skolem theorem.

8.2 Outline of the Proof

The proof of the completeness theorem is a bit complex, and upon first reading it, it is easy to get lost. So let us outline the proof. The first step is a shift of perspective, that allows us to see a route to a proof. When completeness is thought of as “whenever $\Gamma \models A$ then $\Gamma \vdash A$,” it may be hard to even come up with an idea: for to show that $\Gamma \vdash A$ we have to find a derivation, and it does not look like the hypothesis that $\Gamma \models A$ helps us for this in any way. For some proof systems it is possible to directly construct a derivation, but we will take a slightly different tack. The shift in perspective required is this: completeness can also be formulated as: “if Γ is consistent, it has a model.” Perhaps we can use the information in Γ together with the hypothesis that it is consistent to construct a model. After all, we know what kind of model we are looking for: one that is as Γ describes it!

If Γ contains only atomic sentences, it is easy to construct a model for it: for atomic sentences are all of the form $P(a_1, \dots, a_n)$ where the a_i are constant symbols. So all we have to do is come up with a domain $|M|$ and an interpretation for P so that $M \models P(a_1, \dots, a_n)$. But nothing’s easier than that: put $|M| = \mathbb{N}$, $c_i^M = i$, and for every $P(a_1, \dots, a_n) \in \Gamma$, put the tuple $\langle k_1, \dots, k_n \rangle$ into P^M , where k_i is the index of the constant symbol a_i (i.e., $a_i \equiv c_{k_i}$).

Now suppose Γ contains some sentence $\neg B$, with B atomic. We might worry that the construction of M interferes with the possibility of making $\neg B$ true. But here’s where the consistency

of Γ comes in: if $\neg B \in \Gamma$, then $B \notin \Gamma$, or else Γ would be inconsistent. And if $B \notin \Gamma$, then according to our construction of M , $M \not\models B$, so $M \models \neg B$. So far so good.

Now what if Γ contains complex, non-atomic formulas? Say, it contains $A \wedge B$. Then we should proceed as if both A and B were in Γ . And if $A \vee B \in \Gamma$, then we will have to make at least one of them true, i.e., proceed as if one of them was in Γ .

This suggests the following idea: we add additional sentences to Γ so as to (a) keep the resulting set consistent and (b) make sure that for every possible atomic sentence A , either A is in the resulting set, or $\neg A$, and (c) such that, whenever $A \wedge B$ is in the set, so are both A and B , if $A \vee B$ is in the set, at least one of A or B is also, etc. We keep doing this (potentially forever). Call the set of all sentences so added Γ^* . Then our construction above would provide us with a structure for which we could prove, by induction, that all sentences in Γ^* are true in M , and hence also all sentence in Γ since $\Gamma \subseteq \Gamma^*$.

There is one wrinkle in this plan: if $\exists x A(x) \in \Gamma$ we would hope to be able to pick some constant symbol c and add $A(c)$ in this process. But how do we know we can always do that? Perhaps we only have a few constant symbols in our language, and for each one of them we have $\neg B(c) \in \Gamma$. We can't also add $B(c)$, since this would make the set inconsistent, and we wouldn't know whether M has to make $B(c)$ or $\neg B(c)$ true. Moreover, it might happen that Γ contains only sentences in a language that has no constant symbols at all (e.g., the language of set theory).

The solution to this problem is to simply add infinitely many constants at the beginning, plus sentences that connect them with the quantifiers in the right way. (Of course, we have to verify that this cannot introduce an inconsistency.)

Our original construction works well if we only have constant symbols in the atomic sentences. But the language might also contain function symbols. In that case, it might be tricky to find the right functions on \mathbb{N} to assign to these function symbols to make everything work. So here's another trick: instead of using i to interpret c_i , just take the set of constant symbols itself as

the domain. Then M can assign every constant symbol to itself: $c_i^M = c_i$. But why not go all the way: let $|M|$ be all *terms* of the language! If we do this, there is an obvious assignment of functions (that take terms as arguments and have terms as values) to function symbols: we assign to the function symbol f_i^n the function which, given n terms t_1, \dots, t_n as input, produces the term $f_i^n(t_1, \dots, t_n)$ as value.

The last piece of the puzzle is what to do with $=$. The predicate symbol $=$ has a fixed interpretation: $M \models t = t'$ iff $\text{Val}^M(t) = \text{Val}^M(t')$. Now if we set things up so that the value of a term t is t itself, then this structure will make *no* sentence of the form $t = t'$ true unless t and t' are one and the same term. And of course this is a problem, since basically every interesting theory in a language with function symbols will have as theorems sentences $t = t'$ where t and t' are not the same term (e.g., in theories of arithmetic: $(0 + 0) = 0$). To solve this problem, we change the domain of M : instead of using terms as the objects in $|M|$, we use sets of terms, and each set is so that it contains all those terms which the sentences in Γ require to be equal. So, e.g., if Γ is a theory of arithmetic, one of these sets will contain: $0, (0 + 0), (0 \times 0)$, etc. This will be the set we assign to 0 , and it will turn out that this set is also the value of all the terms in it, e.g., also of $(0 + 0)$. Therefore, the sentence $(0 + 0) = 0$ will be true in this revised structure.

8.3 Maximally Consistent Sets of Sentences

Definition 8.1. A set Γ of sentences is *maximally consistent* iff

1. Γ is consistent, and
2. if $\Gamma \subsetneq \Gamma'$, then Γ' is inconsistent.

An alternate definition equivalent to the above is: a set Γ of sentences is *maximally consistent* iff

1. Γ is consistent, and

2. If $\Gamma \cup \{A\}$ is consistent, then $A \in \Gamma$.

In other words, one cannot add sentences not already in Γ to a maximally consistent set Γ without making the resulting larger set inconsistent.

Maximally consistent sets are important in the completeness proof since we can guarantee that every consistent set of sentences Γ is contained in a maximally consistent set Γ^* , and a maximally consistent set contains, for each sentence A , either A or its negation $\neg A$. This is true in particular for atomic sentences, so from a maximally consistent set in a language suitably expanded by constant symbols, we can construct a structure where the interpretation of predicate symbols is defined according to which atomic sentences are in Γ^* . This structure can then be shown to make all sentences in Γ^* (and hence also in Γ) true. The proof of this latter fact requires that $\neg A \in \Gamma^*$ iff $A \notin \Gamma^*$, $(A \vee B) \in \Gamma^*$ iff $A \in \Gamma^*$ or $B \in \Gamma^*$, etc.

Proposition 8.2. *Suppose Γ is maximally consistent. Then:*

1. *If $\Gamma \vdash A$, then $A \in \Gamma$.*
2. *For any A , either $A \in \Gamma$ or $\neg A \in \Gamma$.*
3. *$(A \wedge B) \in \Gamma$ iff both $A \in \Gamma$ and $B \in \Gamma$.*
4. *$(A \vee B) \in \Gamma$ iff either $A \in \Gamma$ or $B \in \Gamma$.*
5. *$(A \rightarrow B) \in \Gamma$ iff either $A \notin \Gamma$ or $B \in \Gamma$.*

Proof. Let us suppose for all of the following that Γ is maximally consistent.

1. If $\Gamma \vdash A$, then $A \in \Gamma$.

Suppose that $\Gamma \vdash A$. Suppose to the contrary that $A \notin \Gamma$: then since Γ is maximally consistent, $\Gamma \cup \{A\}$ is inconsistent, hence $\Gamma \cup \{A\} \vdash \perp$. By Proposition 7.15, Γ is inconsis-

tent. This contradicts the assumption that Γ is consistent. Hence, it cannot be the case that $A \notin \Gamma$, so $A \in \Gamma$.

2. For any A , either $A \in \Gamma$ or $\neg A \in \Gamma$.

Suppose to the contrary that for some A both $A \notin \Gamma$ and $\neg A \notin \Gamma$. Since Γ is maximally consistent, $\Gamma \cup \{A\}$ and $\Gamma \cup \{\neg A\}$ are both inconsistent, so $\Gamma \cup \{A\} \vdash \perp$ and $\Gamma \cup \{\neg A\} \vdash \perp$. By [Proposition 7.17](#), Γ is inconsistent, a contradiction. Hence there cannot be such a sentence A and, for every A , $A \in \Gamma$ or $\neg A \in \Gamma$.

3. Exercise.

4. $(A \vee B) \in \Gamma$ iff either $A \in \Gamma$ or $B \in \Gamma$.

For the contrapositive of the forward direction, suppose that $A \notin \Gamma$ and $B \notin \Gamma$. We want to show that $(A \vee B) \notin \Gamma$. Since Γ is maximally consistent, $\Gamma \cup \{A\} \vdash \perp$ and $\Gamma \cup \{B\} \vdash \perp$. By [Proposition 7.18](#), $\Gamma \cup \{(A \vee B)\}$ is inconsistent. Hence, $(A \vee B) \notin \Gamma$, as required.

For the reverse direction, suppose that $A \in \Gamma$ or $B \in \Gamma$. Then $\Gamma \vdash A$ or $\Gamma \vdash B$. By [Proposition 7.19](#), $\Gamma \vdash A \vee B$. By [\(1\)](#), $(A \vee B) \in \Gamma$, as required.

5. Exercise.

□

8.4 Henkin Expansion

Part of the challenge in proving the completeness theorem is that the model we construct from a maximally consistent set Γ must make all the quantified formulas in Γ true. In order to guarantee this, we use a trick due to Leon Henkin. In essence, the trick consists in expanding the language by infinitely many constants and adding, for each formula with one free variable $A(x)$ a formula of the form $\exists x A \rightarrow A(c)$, where c is one of the new constant

symbols. When we construct the structure satisfying Γ , this will guarantee that each true existential sentence has a witness among the new constants.

Lemma 8.3. *If Γ is consistent in \mathcal{L} and \mathcal{L}' is obtained from \mathcal{L} by adding a countably infinite set of new constant symbols c_1, c_2, \dots , then Γ is consistent in \mathcal{L}' .*

Definition 8.4. A set Γ of formulas of a language \mathcal{L} is *saturated* if and only if for each formula $A \in \text{Frm}(\mathcal{L})$ and variable x there is a constant symbol c such that $\exists x A \rightarrow A(c) \in \Gamma$.

The following definition will be used in the proof of the next theorem.

Definition 8.5. Let \mathcal{L}' be as in Lemma 8.3. Fix an enumeration $\langle A_1, x_1 \rangle, \langle A_2, x_2 \rangle, \dots$ of all formula-variable pairs of \mathcal{L}' . We define the sentences D_n by recursion on n . Assuming that D_1, \dots, D_n have already been defined, let c_{n+1} be the first new constant symbol among the d_i that does not occur in D_1, \dots, D_n , and let D_{n+1} be the formula $\exists x_{n+1} A_{n+1}(x_{n+1}) \rightarrow A_{n+1}(c_{n+1})$. This includes the case where $n = 0$ and the list of previous D_i 's is empty, i.e., D_1 is $\exists x_1 A_1 \rightarrow A_1(c_1)$.

Theorem 8.6. *Every consistent set Γ can be extended to a saturated consistent set Γ' .*

Proof. Given a consistent set of sentences Γ in a language \mathcal{L} , expand the language by adding a countably infinite set of new constant symbols to form \mathcal{L}' . By the previous Lemma, Γ is still consistent in the richer language. Further, let D_i be as in the previous definition: then $\Gamma \cup \{D_1, D_2, \dots\}$ is saturated by construction. Let

$$\begin{aligned}\Gamma_0 &= \Gamma \\ \Gamma_{n+1} &= \Gamma_n \cup \{D_{n+1}\}\end{aligned}$$

i.e., $\Gamma_n = \Gamma \cup \{D_1, \dots, D_n\}$, and let $\Gamma' = \bigcup_n \Gamma_n$. To show that Γ' is consistent it suffices to show, by induction on n , that each set Γ_n is consistent.

The induction basis is simply the claim that $\Gamma_0 = \Gamma$ is consistent, which is the hypothesis of the theorem. For the induction step, suppose that Γ_{n-1} is consistent but $\Gamma_n = \Gamma_{n-1} \cup \{D_n\}$ is inconsistent. Recall that D_n is $\exists x_n A_n(x_n) \rightarrow A_n(c_n)$, where $A(x)$ is a formula of \mathcal{L}' with only the variable x_n free and not containing any constant symbols c_i where $i \geq n$.

If $\Gamma_{n-1} \cup \{D_n\}$ is inconsistent, then $\Gamma_{n-1} \vdash \neg D_n$, and hence both of the following hold:

$$\Gamma_{n-1} \vdash \exists x_n A_n(x_n) \quad \Gamma_{n-1} \vdash \neg A_n(c_n)$$

Here c_n does not occur in Γ_{n-1} or $A_n(x_n)$ (remember, it was added only with D_n). By [Theorem 7.24](#), from $\Gamma \vdash \neg A_n(c_n)$, we obtain $\Gamma \vdash \forall x_n \neg A_n(x_n)$. Thus we have that both $\Gamma_{n-1} \vdash \exists x_n A_n$ and $\Gamma_{n-1} \vdash \forall x_n \neg A_n(x_n)$, so Γ itself is inconsistent. (Note that $\forall x_n \neg A_n(x_n) \vdash \neg \exists x_n A_n(x_n)$.) Contradiction: Γ_{n-1} was supposed to be consistent. Hence $\Gamma_n \cup \{D_n\}$ is consistent. \square

8.5 Lindenbaum's Lemma

Lemma 8.7 (Lindenbaum's Lemma). *Every consistent set Γ can be extended to a maximally consistent saturated set Γ^* .*

Proof. Let Γ be consistent, and let Γ' be as in the proof of [Theorem 8.6](#): we proved there that $\Gamma \cup \Gamma'$ is a consistent saturated set in the richer language \mathcal{L}' (with the countably infinite set of new constants). Let A_0, A_1, \dots be an enumeration of all the formulas of \mathcal{L}' . Define $\Gamma_0 = \Gamma \cup \Gamma'$, and

$$\Gamma_{n+1} = \begin{cases} \Gamma_n \cup \{A_n\} & \text{if } \Gamma_n \cup \{A_n\} \text{ is consistent;} \\ \Gamma_n \cup \{\neg A_n\} & \text{otherwise.} \end{cases}$$

Let $\Gamma^* = \bigcup_{n \geq 0} \Gamma_n$. Since $\Gamma' \subseteq \Gamma^*$, for each formula A , Γ^* contains a formula of the form $\exists x A \rightarrow A(c)$ and thus is saturated.

Each Γ_n is consistent: Γ_0 is consistent by definition. If $\Gamma_{n+1} = \Gamma_n \cup \{A\}$, this is because the latter is consistent. If it isn't, $\Gamma_{n+1} = \Gamma_n \cup \{\neg A\}$, which must be consistent. If it weren't, i.e., both $\Gamma_n \cup \{A\}$ and $\Gamma_n \cup \{\neg A\}$ are inconsistent, then $\Gamma_n \vdash \neg A$ and $\Gamma_n \vdash A$, so Γ_n would be inconsistent contrary to induction hypothesis.

Every formula of $\text{Frm}(\mathcal{L}')$ appears on the list used to define Γ^* . If $A_n \notin \Gamma^*$, then that is because $\Gamma_n \cup \{A_n\}$ was inconsistent. But that means that Γ^* is maximally consistent. \square

8.6 Construction of a Model

We will begin by showing how to construct a structure which satisfies a maximally consistent, saturated set of sentences in a language \mathcal{L} without $=$.

Definition 8.8. Let Γ^* be a maximally consistent, saturated set of sentences in a language \mathcal{L} . The *term model* $M(\Gamma^*)$ of Γ^* is the structure defined as follows:

1. The domain $|M(\Gamma^*)|$ is the set of all closed terms of \mathcal{L} .
2. The interpretation of a constant symbol c is c itself: $c^{M(\Gamma^*)} = c$.
3. The function symbol f is assigned the function

$$f^{M(\Gamma^*)}(t_1, \dots, t_n) = f(\text{Val}^{M(\Gamma^*)}(t_1), \dots, \text{Val}^{M(\Gamma^*)}(t_n))$$

4. If R is an n -place predicate symbol, then $\langle t_1, \dots, t_n \rangle \in R^{M(\Gamma^*)}$ iff $R(t_1, \dots, t_n) \in \Gamma^*$.

Lemma 8.9 (Truth Lemma). *Suppose A does not contain $=$. Then $M(\Gamma^*) \models A$ iff $A \in \Gamma^*$.*

Proof. We prove both directions simultaneously, and by induction on A .

1. $A \equiv R(t_1, \dots, t_n)$: $M(\Gamma^*) \models R(t_1, \dots, t_n)$ iff $\langle t_1, \dots, t_n \rangle \in R^{M(\Gamma^*)}$ (by the definition of satisfaction) iff $R(t_1, \dots, t_n) \in \Gamma^*$ (the construction of $M(\Gamma^*)$).
2. $A \equiv \neg B$: $M(\Gamma^*) \models A$ iff $M(\Gamma^*) \not\models B$ (by definition of satisfaction). By induction hypothesis, $M(\Gamma^*) \not\models B$ iff $B \notin \Gamma^*$. By **Proposition 8.2(2)**, $\neg B \in \Gamma^*$ if $B \notin \Gamma^*$; and $\neg B \notin \Gamma^*$ if $B \in \Gamma^*$ since Γ^* is consistent.
3. $A \equiv B \wedge C$: exercise.
4. $A \equiv B \vee C$: $M(\Gamma^*) \models A$ iff at $M(\Gamma^*) \models B$ or $M(\Gamma^*) \models C$ (by definition of satisfaction) iff $B \in \Gamma^*$ or $C \in \Gamma^*$ (by induction hypothesis). This is the case iff $(B \vee C) \in \Gamma^*$ (by **Proposition 8.2(4)**).
5. $A \equiv B \rightarrow C$: exercise.
6. $A \equiv \forall x B(x)$: exercise.
7. $A \equiv \exists x B(x)$: First suppose that $M(\Gamma^*) \models A$. By the definition of satisfaction, for some variable assignment s , $M(\Gamma^*), s \models B(x)$. The value $s(x)$ is some term $t \in |M(\Gamma^*)|$. Thus, $M(\Gamma^*) \models B(t)$, and by our induction hypothesis, $B(t) \in \Gamma^*$. By **Theorem 7.25** we have $\Gamma^* \vdash \exists x B(x)$. Then, by **Proposition 8.2(1)**, we can conclude that $A \in \Gamma^*$.

Conversely, suppose that $\exists x B(x) \in \Gamma^*$. Because Γ^* is saturated, $(\exists x B(x) \rightarrow B(c)) \in \Gamma^*$. By **Proposition 7.22** together with **Proposition 8.2(1)**, $B(c) \in \Gamma^*$. By inductive hypothesis, $M(\Gamma^*) \models B(c)$. Now consider the variable assignment with $s(x) = c^{M(\Gamma^*)}$. Then $M(\Gamma^*), s \models B(x)$. By definition of satisfaction, $M(\Gamma^*) \models \exists x B(x)$.

□

8.7 Identity

The construction of the term model given in the preceding section is enough to establish completeness for first-order logic for sets Γ that do not contain $=$. The term model satisfies every $A \in \Gamma^*$ which does not contain $=$ (and hence all $A \in \Gamma$). It does not work, however, if $=$ is present. The reason is that Γ^* then may contain a sentence $t = t'$, but in the term model the value of any term is that term itself. Hence, if t and t' are different terms, their values in the term model—i.e., t and t' , respectively—are different, and so $t = t'$ is false. We can fix this, however, using a construction known as “factoring.”

Definition 8.10. Let Γ^* be a maximally consistent set of sentences in \mathcal{L} . We define the relation \approx on the set of closed terms of \mathcal{L} by

$$t \approx t' \quad \text{iff} \quad t = t' \in \Gamma^*$$

Proposition 8.11. *The relation \approx has the following properties:*

1. \approx is reflexive.
2. \approx is symmetric.
3. \approx is transitive.
4. If $t \approx t'$, f is a function symbol, and $t_1, \dots, t_{i-1}, t_{i+1}, \dots, t_n$ are terms, then

$$f(t_1, \dots, t_{i-1}, t, t_{i+1}, \dots, t_n) \approx f(t_1, \dots, t_{i-1}, t', t_{i+1}, \dots, t_n).$$

5. If $t \approx t'$, R is a function symbol, and $t_1, \dots, t_{i-1}, t_{i+1}, \dots, t_n$ are terms, then

$$R(t_1, \dots, t_{i-1}, t, t_{i+1}, \dots, t_n) \in \Gamma^* \quad \text{iff} \quad R(t_1, \dots, t_{i-1}, t', t_{i+1}, \dots, t_n) \in \Gamma^*.$$

Proof. Since Γ^* is maximally consistent, $t = t' \in \Gamma^*$ iff $\Gamma^* \vdash t = t'$. Thus it is enough to show the following:

1. $\Gamma^* \vdash t = t$ for all terms t .
2. If $\Gamma^* \vdash t = t'$ then $\Gamma^* \vdash t' = t$.
3. If $\Gamma^* \vdash t = t'$ and $\Gamma^* \vdash t' = t''$, then $\Gamma^* \vdash t = t''$.
4. If $\Gamma^* \vdash t = t'$, then

$$\Gamma^* \vdash f(t_1, \dots, t_{i-1}, t, t_{i+1}, \dots, t_n) = f(t_1, \dots, t_{i-1}, t', t_{i+1}, \dots, t_n)$$

for every n -place function symbol f and terms $t_1, \dots, t_{i-1}, t_{i+1}, \dots, t_n$.

5. If $\Gamma^* \vdash t = t'$ and $\Gamma^* \vdash R(t_1, \dots, t_{i-1}, t, t_{i+1}, \dots, t_n)$, then $\Gamma^* \vdash R(t_1, \dots, t_{i-1}, t', t_{i+1}, \dots, t_n)$ for every n -place predicate symbol R and terms $t_1, \dots, t_{i-1}, t_{i+1}, \dots, t_n$.

□

Definition 8.12. Suppose Γ^* is a maximally consistent set in a language \mathcal{L} , t is a term, and \approx as in the previous definition. Then:

$$[t]_{\approx} = \{t' : t' \in \text{Trm}(\mathcal{L}), t \approx t'\}$$

and $\text{Trm}(\mathcal{L})/\approx = \{[t]_{\approx} : t \in \text{Trm}(\mathcal{L})\}$.

Definition 8.13. Let $M = M(\Gamma^*)$ be the term model for Γ^* . Then M/\approx is the following structure:

1. $|M/\approx| = \text{Trm}(\mathcal{L})/\approx$.
2. $c^{M/\approx} = [c]_{\approx}$
3. $f^{M/\approx}([t_1]_{\approx}, \dots, [t_n]_{\approx}) = [f(t_1, \dots, t_n)]_{\approx}$
4. $\langle [t_1]_{\approx}, \dots, [t_n]_{\approx} \rangle \in R^{M/\approx}$ iff $M \models R(t_1, \dots, t_n)$.

Note that we have defined $f^{M/\approx}$ and $R^{M/\approx}$ for elements of $\text{Trm}(\mathcal{L})/\approx$ by referring to them as $[t]_\approx$, i.e., via *representatives* $t \in [t]_\approx$. We have to make sure that these definitions do not depend on the choice of these representatives, i.e., that for some other choices t' which determine the same equivalence classes ($[t]_\approx = [t']_\approx$), the definitions yield the same result. For instance, if R is a one-place predicate symbol, the last clause of the definition says that $[t]_\approx \in R^{M/\approx}$ iff $M \models R(t)$. If for some other term t' with $t \approx t'$, $M \not\models R(t)$, then the definition would require $[t']_\approx \notin R^{M/\approx}$. If $t \approx t'$, then $[t]_\approx = [t']_\approx$, but we can't have both $[t]_\approx \in R^{M/\approx}$ and $[t]_\approx \notin R^{M/\approx}$. However, [Proposition 8.11](#) guarantees that this cannot happen.

Proposition 8.14. *M/\approx is well defined, i.e., if $t_1, \dots, t_n, t'_1, \dots, t'_n$ are terms, and $t_i \approx t'_i$ then*

1. $[f(t_1, \dots, t_n)]_\approx = [f(t'_1, \dots, t'_n)]_\approx$, i.e., $f(t_1, \dots, t_n) \approx f(t'_1, \dots, t'_n)$ and
2. $M \models R(t_1, \dots, t_n)$ iff $M \models R(t'_1, \dots, t'_n)$, i.e., $R(t_1, \dots, t_n) \in \Gamma^*$ iff $R(t'_1, \dots, t'_n) \in \Gamma^*$.

Proof. Follows from [Proposition 8.11](#). □

Lemma 8.15. *$M/\approx \models A$ iff $A \in \Gamma^*$ for all sentences A .*

Proof. By induction on A , just as in the proof of [Lemma 8.9](#). The only case that needs additional attention is when $A \equiv t = t'$.

$$\begin{aligned}
 M/\approx \models t = t' & \text{ iff } [t]_\approx = [t']_\approx \text{ (by definition of } M/\approx) \\
 & \text{ iff } t \approx t' \text{ (by definition of } [t]_\approx) \\
 & \text{ iff } t = t' \in \Gamma^* \text{ (by definition of } \approx).
 \end{aligned}$$

□

Note that while $M(\Gamma^*)$ is always countable and infinite, M/\approx may be finite, since it may turn out that there are only finitely many classes $[t]_\approx$. This is to be expected, since Γ may contain sentences which require any structure in which they are true to be finite. For instance, $\forall x \forall y x = y$ is a consistent sentence, but is satisfied only in structures with a domain that contains exactly one element.

8.8 The Completeness Theorem

Theorem 8.16 (Completeness Theorem). *Let Γ be a set of sentences. If Γ is consistent, it is satisfiable.*

Proof. Suppose Γ is consistent. By Lemma 8.7, there is a $\Gamma^* \supseteq \Gamma$ which is maximally consistent and saturated. If Γ does not contain $=$, then by Lemma 8.9, $M(\Gamma^*) \models A$ iff $A \in \Gamma^*$. From this it follows in particular that for all $A \in \Gamma$, $M(\Gamma^*) \models A$, so Γ is satisfiable. If Γ does contain $=$, then by Lemma 8.15, $M/\approx \models A$ iff $A \in \Gamma^*$ for all sentences A . In particular, $M/\approx \models A$ for all $A \in \Gamma$, so Γ is satisfiable. \square

Corollary 8.17 (Completeness Theorem, Second Version). *For all Γ and A sentences: if $\Gamma \models A$ then $\Gamma \vdash A$.*

Proof. Note that the Γ 's in Corollary 8.17 and Theorem 8.16 are universally quantified. To make sure we do not confuse ourselves, let us restate Theorem 8.16 using a different variable: for any set of sentences Δ , if Δ is consistent, it is satisfiable. By contraposition, if Δ is not satisfiable, then Δ is inconsistent. We will use this to prove the corollary.

Suppose that $\Gamma \models A$. Then $\Gamma \cup \{\neg A\}$ is unsatisfiable by Proposition 5.45. Taking $\Gamma \cup \{\neg A\}$ as our Δ , the previous version of Theorem 8.16 gives us that $\Gamma \cup \{\neg A\}$ is inconsistent. By Proposition 7.11, $\Gamma \vdash A$. \square

8.9 The Compactness Theorem

Definition 8.18. A set Γ of formulas is *finitely satisfiable* if and only if every finite $\Gamma_0 \subseteq \Gamma$ is satisfiable.

Theorem 8.19 (Compactness Theorem). *The following hold for any sentences Γ and A :*

1. $\Gamma \models A$ iff there is a finite $\Gamma_0 \subseteq \Gamma$ such that $\Gamma_0 \models A$.
2. Γ is satisfiable if and only if it is finitely satisfiable.

Proof. We prove (2). If Γ is satisfiable, then there is a structure M such that $M \models A$ for all $A \in \Gamma$. Of course, this M also satisfies every finite subset of Γ , so Γ is finitely satisfiable.

Now suppose that Γ is finitely satisfiable. Then every finite subset $\Gamma_0 \subseteq \Gamma$ is satisfiable. By soundness, every finite subset is consistent. Then Γ itself must be consistent. For assume it is not, i.e., $\Gamma \vdash \perp$. But derivations are finite, and so already some finite subset $\Gamma_0 \subseteq \Gamma$ must be inconsistent (cf. [Proposition 7.13](#)). But we just showed they are all consistent, a contradiction. Now by completeness, since Γ is consistent, it is satisfiable. \square

8.10 The Löwenheim-Skolem Theorem

Theorem 8.20. *If Γ is consistent then it has a countably infinite model, i.e., it is satisfiable in a structure whose domain is either finite or infinite but countable.*

Proof. If Γ is consistent, the structure M delivered by the proof of the completeness theorem has a domain $|M|$ whose cardinality is bounded by that of the set of the terms of the language \mathcal{L} . So M is at most countably infinite. \square

Theorem 8.21. *If Γ is consistent set of sentences in the language of first-order logic without identity, then it has a countably infinite model, i.e., it is satisfiable in a structure whose domain is infinite and countable.*

Proof. If Γ is consistent and contains no sentences in which identity appears, then the structure M delivered by the proof of the completeness theorem has a domain $|M|$ whose cardinality is identical to that of the set of the terms of the language \mathcal{L} . So M is denumerably infinite. \square



CHAPTER 9

Beyond First-order Logic

9.1 Overview

First-order logic is not the only system of logic of interest: there are many extensions and variations of first-order logic. A logic typically consists of the formal specification of a language, usually, but not always, a deductive system, and usually, but not always, an intended semantics. But the technical use of the term raises an obvious question: what do logics that are not first-order logic have to do with the word “logic,” used in the intuitive or philosophical sense? All of the systems described below are designed to model reasoning of some form or another; can we say what makes them logical?

No easy answers are forthcoming. The word “logic” is used in different ways and in different contexts, and the notion, like that of “truth,” has been analyzed from numerous philosophical stances. For example, one might take the goal of logical reason-

ing to be the determination of which statements are necessarily true, true a priori, true independent of the interpretation of the nonlogical terms, true by virtue of their form, or true by linguistic convention; and each of these conceptions requires a good deal of clarification. Even if one restricts one's attention to the kind of logic used in mathematical, there is little agreement as to its scope. For example, in the *Principia Mathematica*, Russell and Whitehead tried to develop mathematics on the basis of logic, in the *logicist* tradition begun by Frege. Their system of logic was a form of higher-type logic similar to the one described below. In the end they were forced to introduce axioms which, by most standards, do not seem purely logical (notably, the axiom of infinity, and the axiom of reducibility), but one might nonetheless hold that some forms of higher-order reasoning should be accepted as logical. In contrast, Quine, whose ontology does not admit "propositions" as legitimate objects of discourse, argues that second-order and higher-order logic are really manifestations of set theory in sheep's clothing; in other words, systems involving quantification over predicates are not purely logical.

For now, it is best to leave such philosophical issues for a rainy day, and simply think of the systems below as formal idealizations of various kinds of reasoning, logical or otherwise.

9.2 Many-Sorted Logic

In first-order logic, variables and quantifiers range over a single domain. But it is often useful to have multiple (disjoint) domains: for example, you might want to have a domain of numbers, a domain of geometric objects, a domain of functions from numbers to numbers, a domain of abelian groups, and so on.

Many-sorted logic provides this kind of framework. One starts with a list of "sorts"—the "sort" of an object indicates the "domain" it is supposed to inhabit. One then has variables and quantifiers for each sort, and (usually) an identity predicate for each sort. Functions and relations are also "typed" by the sorts of ob-

jects they can take as arguments. Otherwise, one keeps the usual rules of first-order logic, with versions of the quantifier-rules repeated for each sort.

For example, to study international relations we might choose a language with two sorts of objects, French citizens and German citizens. We might have a unary relation, “drinks wine,” for objects of the first sort; another unary relation, “eats wurst,” for objects of the second sort; and a binary relation, “forms a multinational married couple,” which takes two arguments, where the first argument is of the first sort and the second argument is of the second sort. If we use variables a, b, c to range over French citizens and x, y, z to range over German citizens, then

$$\forall a \forall x [(MarriedTo(a, x) \rightarrow (DrinksWine(a) \vee \neg EatsWurst(x)))]$$

asserts that if any French person is married to a German, either the French person drinks wine or the German doesn’t eat wurst.

Many-sorted logic can be embedded in first-order logic in a natural way, by lumping all the objects of the many-sorted domains together into one first-order domain, using unary predicate symbols to keep track of the sorts, and relativizing quantifiers. For example, the first-order language corresponding to the example above would have unary predicate symbols “*German*” and “*French*,” in addition to the other relations described, with the sort requirements erased. A sorted quantifier $\forall x A$, where x is a variable of the German sort, translates to

$$\forall x (German(x) \rightarrow A).$$

We need to add axioms that insure that the sorts are separate—e.g., $\forall x \neg (German(x) \wedge French(x))$ —as well as axioms that guarantee that “drinks wine” only holds of objects satisfying the predicate *French*(x), etc. With these conventions and axioms, it is not difficult to show that many-sorted sentences translate to first-order sentences, and many-sorted derivations translate to first-order derivations. Also, many-sorted structures “translate” to corresponding first-order structures and vice-versa, so we also have a completeness theorem for many-sorted logic.

9.3 Second-Order logic

The language of second-order logic allows one to quantify not just over a domain of individuals, but over relations on that domain as well. Given a first-order language \mathcal{L} , for each k one adds variables R which range over k -ary relations, and allows quantification over those variables. If R is a variable for a k -ary relation, and t_1, \dots, t_k are ordinary (first-order) terms, $R(t_1, \dots, t_k)$ is an atomic formula. Otherwise, the set of formulas is defined just as in the case of first-order logic, with additional clauses for second-order quantification. Note that we only have the identity predicate for first-order terms: if R and S are relation variables of the same arity k , we can define $R = S$ to be an abbreviation for

$$\forall x_1 \dots \forall x_k (R(x_1, \dots, x_k) \leftrightarrow S(x_1, \dots, x_k)).$$

The rules for second-order logic simply extend the quantifier rules to the new second order variables. Here, however, one has to be a little bit careful to explain how these variables interact with the predicate symbols of \mathcal{L} , and with formulas of \mathcal{L} more generally. At the bare minimum, relation variables count as terms, so one has inferences of the form

$$A(R) \vdash \exists R A(R)$$

But if \mathcal{L} is the language of arithmetic with a constant relation symbol $<$, one would also expect the following inference to be valid:

$$x < y \vdash \exists R R(x, y)$$

or for a given formula A ,

$$A(x_1, \dots, x_k) \vdash \exists R R(x_1, \dots, x_k)$$

More generally, we might want to allow inferences of the form

$$A[\lambda \vec{x}. B(\vec{x})/R] \vdash \exists R A$$

where $A[\lambda \vec{x}. B(\vec{x})/R]$ denotes the result of replacing every atomic formula of the form Rt_1, \dots, t_k in A by $B(t_1, \dots, t_k)$. This last rule

is equivalent to having a *comprehension schema*, i.e., an axiom of the form

$$\exists R \forall x_1, \dots, x_k (A(x_1, \dots, x_k) \leftrightarrow R(x_1, \dots, x_k)),$$

one for each formula A in the second-order language, in which R is not a free variable. (Exercise: show that if R is allowed to occur in A , this schema is inconsistent!)

When logicians refer to the “axioms of second-order logic” they usually mean the minimal extension of first-order logic by second-order quantifier rules together with the comprehension schema. But it is often interesting to study weaker subsystems of these axioms and rules. For example, note that in its full generality the axiom schema of comprehension is *impredicative*: it allows one to assert the existence of a relation $R(x_1, \dots, x_k)$ that is “defined” by a formula with second-order quantifiers; and these quantifiers range over the set of all such relations—a set which includes R itself! Around the turn of the twentieth century, a common reaction to Russell’s paradox was to lay the blame on such definitions, and to avoid them in developing the foundations of mathematics. If one prohibits the use of second-order quantifiers in the formula A , one has a *predicative* form of comprehension, which is somewhat weaker.

From the semantic point of view, one can think of a second-order structure as consisting of a first-order structure for the language, coupled with a set of relations on the domain over which the second-order quantifiers range (more precisely, for each k there is a set of relations of arity k). Of course, if comprehension is included in the proof system, then we have the added requirement that there are enough relations in the “second-order part” to satisfy the comprehension axioms—otherwise the proof system is not sound! One easy way to insure that there are enough relations around is to take the second-order part to consist of *all* the relations on the first-order part. Such a structure is called *full*, and, in a sense, is really the “intended structure” for the language. If we restrict our attention to full structures we have what

is known as the *full* second-order semantics. In that case, specifying a structure boils down to specifying the first-order part, since the contents of the second-order part follow from that implicitly.

To summarize, there is some ambiguity when talking about second-order logic. In terms of the proof system, one might have in mind either

1. A “minimal” second-order proof system, together with some comprehension axioms.
2. The “standard” second-order proof system, with full comprehension.

In terms of the semantics, one might be interested in either

1. The “weak” semantics, where a structure consists of a first-order part, together with a second-order part big enough to satisfy the comprehension axioms.
2. The “standard” second-order semantics, in which one considers full structures only.

When logicians do not specify the proof system or the semantics they have in mind, they are usually referring to the second item on each list. The advantage to using this semantics is that, as we will see, it gives us categorical descriptions of many natural mathematical structures; at the same time, the proof system is quite strong, and sound for this semantics. The drawback is that the proof system is *not* complete for the semantics; in fact, *no* effectively given proof system is complete for the full second-order semantics. On the other hand, we will see that the proof system *is* complete for the weakened semantics; this implies that if a sentence is not provable, then there is *some* structure, not necessarily the full one, in which it is false.

The language of second-order logic is quite rich. One can identify unary relations with subsets of the domain, and so in

particular you can quantify over these sets; for example, one can express induction for the natural numbers with a single axiom

$$\forall R ((R(0) \wedge \forall x (R(x) \rightarrow R(x')))) \rightarrow \forall x R(x)).$$

If one takes the language of arithmetic to have symbols $0, \iota, +, \times$ and $<$, one can add the following axioms to describe their behavior:

1. $\forall x \neg x' = 0$
2. $\forall x \forall y (s(x) = s(y) \rightarrow x = y)$
3. $\forall x (x + 0) = x$
4. $\forall x \forall y (x + y') = (x + y)'$
5. $\forall x (x \times 0) = 0$
6. $\forall x \forall y (x \times y') = ((x \times y) + x)$
7. $\forall x \forall y (x < y \leftrightarrow \exists z y = (x + z'))$

It is not difficult to show that these axioms, together with the axiom of induction above, provide a categorical description of the structure \mathbb{N} , the standard model of arithmetic, provided we are using the full second-order semantics. Given any structure A in which these axioms are true, define a function f from \mathbb{N} to the domain of A using ordinary recursion on \mathbb{N} , so that $f(0) = 0^A$ and $f(x + 1) = \iota^A(f(x))$. Using ordinary induction on \mathbb{N} and the fact that axioms (1) and (2) hold in A , we see that f is injective. To see that f is surjective, let P be the set of elements of $|A|$ that are in the range of f . Since A is full, P is in the second-order domain. By the construction of f , we know that 0^A is in P , and that P is closed under ι^A . The fact that the induction axiom holds in A (in particular, for P) guarantees that P is equal to the entire first-order domain of A . This shows that f is a bijection. Showing that f is a homomorphism is no more difficult, using ordinary induction on \mathbb{N} repeatedly.

In set-theoretic terms, a function is just a special kind of relation; for example, a unary function f can be identified with a binary relation R satisfying $\forall x \exists y R(x, y)$. As a result, one can quantify over functions too. Using the full semantics, one can then define the class of infinite structures to be the class of structures A for which there is an injective function from the domain of A to a proper subset of itself:

$$\exists f (\forall x \forall y (f(x) = f(y) \rightarrow x = y) \wedge \exists y \forall x f(x) \neq y).$$

The negation of this sentence then defines the class of finite structures.

In addition, one can define the class of well-orderings, by adding the following to the definition of a linear ordering:

$$\forall P (\exists x P(x) \rightarrow \exists x (P(x) \wedge \forall y (y < x \rightarrow \neg P(y)))).$$

This asserts that every nonempty set has a least element, modulo the identification of “set” with “one-place relation”. For another example, one can express the notion of connectedness for graphs, by saying that there is no nontrivial separation of the vertices into disconnected parts:

$$\neg \exists A (\exists x A(x) \wedge \exists y \neg A(y) \wedge \forall [w][\forall z ((A(w) \wedge \neg A(z)) \rightarrow \neg R(w, z))]).$$

For yet another example, you might try as an exercise to define the class of finite structures whose domain has even size. More strikingly, one can provide a categorical description of the real numbers as a complete ordered field containing the rationals.

In short, second-order logic is much more expressive than first-order logic. That’s the good news; now for the bad. We have already mentioned that there is no effective proof system that is complete for the full second-order semantics. For better or for worse, many of the properties of first-order logic are absent, including compactness and the Löwenheim-Skolem theorems.

On the other hand, if one is willing to give up the full second-order semantics in terms of the weaker one, then the minimal

second-order proof system is complete for this semantics. In other words, if we read \vdash as “proves in the minimal system” and \models as “logically implies in the weaker semantics”, we can show that whenever $\Gamma \models A$ then $\Gamma \vdash A$. If one wants to include specific comprehension axioms in the proof system, one has to restrict the semantics to second-order structures that satisfy these axioms: for example, if Δ consists of a set of comprehension axioms (possibly all of them), we have that if $\Gamma \cup \Delta \models A$, then $\Gamma \cup \Delta \vdash A$. In particular, if A is not provable using the comprehension axioms we are considering, then there is a model of $\neg A$ in which these comprehension axioms nonetheless hold.

The easiest way to see that the completeness theorem holds for the weaker semantics is to think of second-order logic as a many-sorted logic, as follows. One sort is interpreted as the ordinary “first-order” domain, and then for each k we have a domain of “relations of arity k .” We take the language to have built-in relation symbols “ $true_k(R, x_1, \dots, x_k)$ ” which is meant to assert that R holds of x_1, \dots, x_k , where R is a variable of the sort “ k -ary relation” and x_1, \dots, x_k are objects of the first-order sort.

With this identification, the weak second-order semantics is essentially the usual semantics for many-sorted logic; and we have already observed that many-sorted logic can be embedded in first-order logic. Modulo the translations back and forth, then, the weaker conception of second-order logic is really a form of first-order logic in disguise, where the domain contains both “objects” and “relations” governed by the appropriate axioms.

9.4 Higher-Order logic

Passing from first-order logic to second-order logic enabled us to talk about sets of objects in the first-order domain, within the formal language. Why stop there? For example, third-order logic should enable us to deal with sets of sets of objects, or perhaps even sets which contain both objects and sets of objects. And fourth-order logic will let us talk about sets of objects of that kind.

As you may have guessed, one can iterate this idea arbitrarily.

In practice, higher-order logic is often formulated in terms of functions instead of relations. (Modulo the natural identifications, this difference is inessential.) Given some basic “sorts” A , B , C , \dots (which we will now call “types”), we can create new ones by stipulating

If σ and τ are finite types then so is $\sigma \rightarrow \tau$.

Think of types as syntactic “labels,” which classify the objects we want in our domain; $\sigma \rightarrow \tau$ describes those objects that are functions which take objects of type σ to objects of type τ . For example, we might want to have a type Ω of truth values, “true” and “false,” and a type \mathbb{N} of natural numbers. In that case, you can think of objects of type $\mathbb{N} \rightarrow \Omega$ as unary relations, or subsets of \mathbb{N} ; objects of type $\mathbb{N} \rightarrow \mathbb{N}$ are functions from natural numbers to natural numbers; and objects of type $(\mathbb{N} \rightarrow \mathbb{N}) \rightarrow \mathbb{N}$ are “functionals,” that is, higher-type functions that take functions to numbers.

As in the case of second-order logic, one can think of higher-order logic as a kind of many-sorted logic, where there is a sort for each type of object we want to consider. But it is usually clearer just to define the syntax of higher-type logic from the ground up. For example, we can define a set of finite types inductively, as follows:

1. \mathbb{N} is a finite type.
2. If σ and τ are finite types, then so is $\sigma \rightarrow \tau$.
3. If σ and τ are finite types, so is $\sigma \times \tau$.

Intuitively, \mathbb{N} denotes the type of the natural numbers, $\sigma \rightarrow \tau$ denotes the type of functions from σ to τ , and $\sigma \times \tau$ denotes the type of pairs of objects, one from σ and one from τ . We can then define a set of terms inductively, as follows:

1. For each type σ , there is a stock of variables x, y, z, \dots of type σ

2. 0 is a term of type \mathbb{N}
3. S (successor) is a term of type $\mathbb{N} \rightarrow \mathbb{N}$
4. If s is a term of type σ , and t is a term of type $\mathbb{N} \rightarrow (\sigma \rightarrow \sigma)$, then R_{st} is a term of type $\mathbb{N} \rightarrow \sigma$
5. If s is a term of type $\tau \rightarrow \sigma$ and t is a term of type τ , then $s(t)$ is a term of type σ
6. If s is a term of type σ and x is a variable of type τ , then $\lambda x. s$ is a term of type $\tau \rightarrow \sigma$.
7. If s is a term of type σ and t is a term of type τ , then $\langle s, t \rangle$ is a term of type $\sigma \times \tau$.
8. If s is a term of type $\sigma \times \tau$ then $p_1(s)$ is a term of type σ and $p_2(s)$ is a term of type τ .

Intuitively, R_{st} denotes the function defined recursively by

$$\begin{aligned} R_{st}(0) &= s \\ R_{st}(x+1) &= t(x, R_{st}(x)), \end{aligned}$$

$\langle s, t \rangle$ denotes the pair whose first component is s and whose second component is t , and $p_1(s)$ and $p_2(s)$ denote the first and second elements (“projections”) of s . Finally, $\lambda x. s$ denotes the function f defined by

$$f(x) = s$$

for any x of type σ ; so item (6) gives us a form of comprehension, enabling us to define functions using terms. Formulas are built up from identity predicate statements $s = t$ between terms of the same type, the usual propositional connectives, and higher-type quantification. One can then take the axioms of the system to be the basic equations governing the terms defined above, together with the usual rules of logic with quantifiers and identity predicate.

If one augments the finite type system with a type Ω of truth values, one has to include axioms which govern its use as well. In fact, if one is clever, one can get rid of complex formulas entirely, replacing them with terms of type Ω ! The proof system can then be modified accordingly. The result is essentially the *simple theory of types* set forth by Alonzo Church in the 1930s.

As in the case of second-order logic, there are different versions of higher-type semantics that one might want to use. In the full version, variables of type $\sigma \rightarrow \tau$ range over the set of *all* functions from the objects of type σ to objects of type τ . As you might expect, this semantics is too strong to admit a complete, effective proof system. But one can consider a weaker semantics, in which a structure consists of sets of elements T_τ for each type τ , together with appropriate operations for application, projection, etc. If the details are carried out correctly, one can obtain completeness theorems for the kinds of proof systems described above.

Higher-type logic is attractive because it provides a framework in which we can embed a good deal of mathematics in a natural way: starting with \mathbb{N} , one can define real numbers, continuous functions, and so on. It is also particularly attractive in the context of intuitionistic logic, since the types have clear “constructive” interpretations. In fact, one can develop constructive versions of higher-type semantics (based on intuitionistic, rather than classical logic) that clarify these constructive interpretations quite nicely, and are, in many ways, more interesting than the classical counterparts.

9.5 Intuitionistic logic

In contrast to second-order and higher-order logic, intuitionistic first-order logic represents a restriction of the classical version, intended to model a more “constructive” kind of reasoning. The following examples may serve to illustrate some of the underlying motivations.

Suppose someone came up to you one day and announced that they had determined a natural number x , with the property that if x is prime, the Riemann hypothesis is true, and if x is composite, the Riemann hypothesis is false. Great news! Whether the Riemann hypothesis is true or not is one of the big open questions of mathematics, and here they seem to have reduced the problem to one of calculation, that is, to the determination of whether a specific number is prime or not.

What is the magic value of x ? They describe it as follows: x is the natural number that is equal to 7 if the Riemann hypothesis is true, and 9 otherwise.

Angrily, you demand your money back. From a classical point of view, the description above does in fact determine a unique value of x ; but what you really want is a value of x that is given *explicitly*.

To take another, perhaps less contrived example, consider the following question. We know that it is possible to raise an irrational number to a rational power, and get a rational result. For example, $\sqrt{2}^2 = 2$. What is less clear is whether or not it is possible to raise an irrational number to an *irrational* power, and get a rational result. The following theorem answers this in the affirmative:

Theorem 9.1. *There are irrational numbers a and b such that a^b is rational.*

Proof. Consider $\sqrt{2}^{\sqrt{2}}$. If this is rational, we are done: we can let $a = b = \sqrt{2}$. Otherwise, it is irrational. Then we have

$$(\sqrt{2}^{\sqrt{2}})^{\sqrt{2}} = \sqrt{2}^{\sqrt{2} \cdot \sqrt{2}} = \sqrt{2}^2 = 2,$$

which is certainly rational. So, in this case, let a be $\sqrt{2}^{\sqrt{2}}$, and let b be $\sqrt{2}$. □

Does this constitute a valid proof? Most mathematicians feel that it does. But again, there is something a little bit unsatisfying here: we have proved the existence of a pair of real numbers with a certain property, without being able to say *which* pair of numbers it is. It is possible to prove the same result, but in such a way that the pair a, b is given in the proof: take $a = \sqrt{3}$ and $b = \log_3 4$. Then

$$a^b = \sqrt{3}^{\log_3 4} = 3^{1/2 \cdot \log_3 4} = (3^{\log_3 4})^{1/2} = 4^{1/2} = 2,$$

since $3^{\log_3 x} = x$.

Intuitionistic logic is designed to model a kind of reasoning where moves like the one in the first proof are disallowed. Proving the existence of an x satisfying $A(x)$ means that you have to give a specific x , and a proof that it satisfies A , like in the second proof. Proving that A or B holds requires that you can prove one or the other.

Formally speaking, intuitionistic first-order logic is what you get if you omit restrict a proof system for first-order logic in a certain way. Similarly, there are intuitionistic versions of second-order or higher-order logic. From the mathematical point of view, these are just formal deductive systems, but, as already noted, they are intended to model a kind of mathematical reasoning. One can take this to be the kind of reasoning that is justified on a certain philosophical view of mathematics (such as Brouwer's intuitionism); one can take it to be a kind of mathematical reasoning which is more "concrete" and satisfying (along the lines of Bishop's constructivism); and one can argue about whether or not the formal description captures the informal motivation. But whatever philosophical positions we may hold, we can study intuitionistic logic as a formally presented logic; and for whatever reasons, many mathematical logicians find it interesting to do so.

There is an informal constructive interpretation of the intuitionist connectives, usually known as the Brouwer-Heyting-Kolmogorov interpretation. It runs as follows: a proof of $A \wedge B$ consists of a proof of A paired with a proof of B ; a proof of $A \vee B$ consists

of either a proof of A , or a proof of B , where we have explicit information as to which is the case; a proof of $A \rightarrow B$ consists of a procedure, which transforms a proof of A to a proof of B ; a proof of $\forall x A(x)$ consists of a procedure which returns a proof of $A(x)$ for any value of x ; and a proof of $\exists x A(x)$ consists of a value of x , together with a proof that this value satisfies A . One can describe the interpretation in computational terms known as the “Curry-Howard isomorphism” or the “formulas-as-types paradigm”: think of a formula as specifying a certain kind of data type, and proofs as computational objects of these data types that enable us to see that the corresponding formula is true.

Intuitionistic logic is often thought of as being classical logic “minus” the law of the excluded middle. This following theorem makes this more precise.

Theorem 9.2. *Intuitionistically, the following axiom schemata are equivalent:*

1. $(A \rightarrow \perp) \rightarrow \neg A$.
2. $A \vee \neg A$
3. $\neg\neg A \rightarrow A$

Obtaining instances of one schema from either of the others is a good exercise in intuitionistic logic.

The first deductive systems for intuitionistic propositional logic, put forth as formalizations of Brouwer’s intuitionism, are due, independently, to Kolmogorov, Glivenko, and Heyting. The first formalization of intuitionistic first-order logic (and parts of intuitionist mathematics) is due to Heyting. Though a number of classically valid schemata are not intuitionistically valid, many are.

The *double-negation translation* describes an important relationship between classical and intuitionist logic. It is defined inductively follows (think of A^N as the “intuitionist” translation of

the classical formula A):

$$\begin{aligned}
 A^N &\equiv \neg\neg A \quad \text{for atomic formulas } A \\
 (A \wedge B)^N &\equiv (A^N \wedge B^N) \\
 (A \vee B)^N &\equiv \neg\neg(A^N \vee B^N) \\
 (A \rightarrow B)^N &\equiv (A^N \rightarrow B^N) \\
 (\forall x A)^N &\equiv \forall x A^N \\
 (\exists x A)^N &\equiv \neg\neg\exists x A^N
 \end{aligned}$$

Kolmogorov and Glivenko had versions of this translation for propositional logic; for predicate logic, it is due to Gödel and Gentzen, independently. We have

Theorem 9.3. 1. $A \leftrightarrow A^N$ is provable classically
 2. If A is provable classically, then A^N is provable intuitionistically.

We can now envision the following dialogue. Classical mathematician: “I’ve proved A !” Intuitionist mathematician: “Your proof isn’t valid. What you’ve really proved is A^N .” Classical mathematician: “Fine by me!” As far as the classical mathematician is concerned, the intuitionist is just splitting hairs, since the two are equivalent. But the intuitionist insists there is a difference.

Note that the above translation concerns pure logic only; it does not address the question as to what the appropriate *nonlogical* axioms are for classical and intuitionistic mathematics, or what the relationship is between them. But the following slight extension of the theorem above provides some useful information:

Theorem 9.4. If Γ proves A classically, Γ^N proves A^N intuitionistically.

In other words, if A is provable from some hypotheses classically, then A^N is provable from their double-negation translations.

To show that a sentence or propositional formula is intuitionistically valid, all you have to do is provide a proof. But how can you show that it is not valid? For that purpose, we need a semantics that is sound, and preferably complete. A semantics due to Kripke nicely fits the bill.

We can play the same game we did for classical logic: define the semantics, and prove soundness and completeness. It is worthwhile, however, to note the following distinction. In the case of classical logic, the semantics was the “obvious” one, in a sense implicit in the meaning of the connectives. Though one can provide some intuitive motivation for Kripke semantics, the latter does not offer the same feeling of inevitability. In addition, the notion of a classical structure is a natural mathematical one, so we can either take the notion of a structure to be a tool for studying classical first-order logic, or take classical first-order logic to be a tool for studying mathematical structures. In contrast, Kripke structures can only be viewed as a logical construct; they don’t seem to have independent mathematical interest.

A Kripke structure for a propositional language consists of a partial order $\text{Mod}(P)$ with a least element, and an “monotone” assignment of propositional variables to the elements of $\text{Mod}(P)$. The intuition is that the elements of $\text{Mod}(P)$ represent “worlds,” or “states of knowledge”; an element $p \geq q$ represents a “possible future state” of q ; and the propositional variables assigned to p are the propositions that are known to be true in state p . The forcing relation $P, p \Vdash A$ then extends this relationship to arbitrary formulas in the language; read $P, p \Vdash A$ as “ A is true in state p .” The relationship is defined inductively, as follows:

1. $P, p \Vdash p_i$ iff p_i is one of the propositional variables assigned to p .
2. $P, p \nVdash \perp$.
3. $P, p \Vdash (A \wedge B)$ iff $P, p \Vdash A$ and $P, p \Vdash B$.

4. $P, p \Vdash (A \vee B)$ iff $P, p \Vdash A$ or $P, p \Vdash B$.
5. $P, p \Vdash (A \rightarrow B)$ iff, whenever $q \geq p$ and $P, q \Vdash A$, then $P, q \Vdash B$.

It is a good exercise to try to show that $\neg(p \wedge q) \rightarrow (\neg p \vee \neg q)$ is not intuitionistically valid, by cooking up a Kripke structure that provides a counterexample.

9.6 Modal Logics

Consider the following example of a conditional sentence:

If Jeremy is alone in that room, then he is drunk and naked and dancing on the chairs.

This is an example of a conditional assertion that may be materially true but nonetheless misleading, since it seems to suggest that there is a stronger link between the antecedent and conclusion other than simply that either the antecedent is false or the consequent true. That is, the wording suggests that the claim is not only true in this particular world (where it may be trivially true, because Jeremy is not alone in the room), but that, moreover, the conclusion *would have* been true *had* the antecedent been true. In other words, one can take the assertion to mean that the claim is true not just in this world, but in any “possible” world; or that it is *necessarily* true, as opposed to just true in this particular world.

Modal logic was designed to make sense of this kind of necessity. One obtains modal propositional logic from ordinary propositional logic by adding a box operator; which is to say, if A is a formula, so is $\Box A$. Intuitively, $\Box A$ asserts that A is *necessarily* true, or true in any possible world. $\Diamond A$ is usually taken to be an abbreviation for $\neg\Box\neg A$, and can be read as asserting that A is *possibly* true. Of course, modality can be added to predicate logic as well.

Kripke structures can be used to provide a semantics for modal logic; in fact, Kripke first designed this semantics with modal logic in mind. Rather than restricting to partial orders, more generally one has a set of “possible worlds,” P , and a binary “accessibility” relation $R(x, y)$ between worlds. Intuitively, $R(p, q)$ asserts that the world q is compatible with p ; i.e., if we are “in” world p , we have to entertain the possibility that the world could have been like q .

Modal logic is sometimes called an “intensional” logic, as opposed to an “extensional” one. The intended semantics for an extensional logic, like classical logic, will only refer to a single world, the “actual” one; while the semantics for an “intensional” logic relies on a more elaborate ontology. In addition to structuring necessity, one can use modality to structure other linguistic constructions, reinterpreting \Box and \Diamond according to the application. For example:

1. In provability logic, $\Box A$ is read “ A is provable” and $\Diamond A$ is read “ A is consistent.”
2. In epistemic logic, one might read $\Box A$ as “I know A ” or “I believe A .”
3. In temporal logic, one can read $\Box A$ as “ A is always true” and $\Diamond A$ as “ A is sometimes true.”

One would like to augment logic with rules and axioms dealing with modality. For example, the system **S4** consists of the ordinary axioms and rules of propositional logic, together with the following axioms:

$$\Box(A \rightarrow B) \rightarrow (\Box A \rightarrow \Box B)$$

$$\Box A \rightarrow A$$

$$\Box A \rightarrow \Box \Box A$$

as well as a rule, “from A conclude $\Box A$.” **S5** adds the following axiom:

$$\Diamond A \rightarrow \Box \Diamond A$$

Variations of these axioms may be suitable for different applications; for example, **S5** is usually taken to characterize the notion of logical necessity. And the nice thing is that one can usually find a semantics for which the proof system is sound and complete by restricting the accessibility relation in the Kripke structures in natural ways. For example, **S4** corresponds to the class of Kripke structures in which the accessibility relation is reflexive and transitive. **S5** corresponds to the class of Kripke structures in which the accessibility relation is *universal*, which is to say that every world is accessible from every other; so $\Box A$ holds if and only if A holds in every world.

9.7 Other Logics

As you may have gathered by now, it is not hard to design a new logic. You too can create your own a syntax, make up a deductive system, and fashion a semantics to go with it. You might have to be a bit clever if you want the proof system to be complete for the semantics, and it might take some effort to convince the world at large that your logic is truly interesting. But, in return, you can enjoy hours of good, clean fun, exploring your logic’s mathematical and computational properties.

Recent decades have witnessed a veritable explosion of formal logics. Fuzzy logic is designed to model reasoning about vague properties. Probabilistic logic is designed to model reasoning about uncertainty. Default logics and nonmonotonic logics are designed to model defeasible forms of reasoning, which is to say, “reasonable” inferences that can later be overturned in the face of new information. There are epistemic logics, designed to model reasoning about knowledge; causal logics, designed to model reasoning about causal relationships; and even “deontic”

logics, which are designed to model reasoning about moral and ethical obligations. Depending on whether the primary motivation for introducing these systems is philosophical, mathematical, or computational, you may find such creatures studies under the rubric of mathematical logic, philosophical logic, artificial intelligence, cognitive science, or elsewhere.

The list goes on and on, and the possibilities seem endless. We may never attain Leibniz' dream of reducing all of human reason to calculation—but that can't stop us from trying.



Alan Turing

1912 - 1954

PART III

Turing Machines

CHAPTER 10

Turing Machine Computations

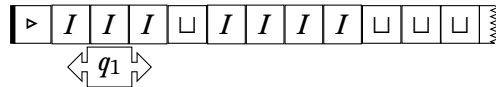
10.1 Introduction

What does it mean for a function, say, from \mathbb{N} to \mathbb{N} to be *computable*? Among the first answers, and the most well known one, is that a function is computable if it can be computed by a Turing machine. This notion was set out by Alan Turing in 1936. Turing machines are an example of *a model of computation*—they are a mathematically precise way of defining the idea of a “computational procedure.” What exactly that means is debated, but it is widely agreed that Turing machines are one way of specifying computational procedures. Even though the term “Turing machine” evokes the image of a physical machine with moving parts, strictly speaking a Turing machine is a purely mathematical construct, and as such it idealizes the idea of a computational procedure. For instance, we place no restriction on either the time or memory requirements of a Turing machine: Turing

machines can compute something even if the computation would require more storage space or more steps than there are atoms in the universe.

It is perhaps best to think of a Turing machine as a program for a special kind of imaginary mechanism. This mechanism consists of a *tape* and a *read-write head*. In our version of Turing machines, the tape is infinite in one direction (to the right), and it is divided into *squares*, each of which may contain a symbol from a finite *alphabet*. Such alphabets can contain any number of different symbols, but we will mainly make do with three: \triangleright , \sqcup , and I . When the mechanism is started, the tape is empty (i.e., each square contains the symbol \sqcup) except for the leftmost square, which contains \triangleright , and a finite number of squares which contain the *input*. At any time, the mechanism is in one of a finite number of *states*. At the outset, the head scans the leftmost square and in a specified *initial state*. At each step of the mechanism's run, the content of the square currently scanned together with the state the mechanism is in and the Turing machine program determine what happens next. The Turing machine program is given by a partial function which takes as input a state q and a symbol σ and outputs a triple $\langle q', \sigma', D \rangle$. Whenever the mechanism is in state q and reads symbol σ , it replaces the symbol on the current square with σ' , the head moves left, right, or stays put according to whether D is L , R , or N , and the mechanism goes into state q' .

For instance, consider the situation below:



The tape of the Turing machine contains the end-of-tape symbol \triangleright on the leftmost square, followed by three I 's, a \sqcup , four more I 's, and the rest of the tape is filled with \sqcup 's. The head is reading the third square from the left, which contains a I , and is in state q_1 —we say “the machine is reading a I in state q_1 .” If the program of the Turing machine returns, for input $\langle q_1, I \rangle$, the triple $\langle q_5, \sqcup, R \rangle$, we would now replace the I on the third square

with a \sqcup , move right to the fourth square, and change the state of the machine to q_5 .

We say that the machine *halts* when it encounters some state, q_n , and symbol, σ such that there is no instruction for $\langle q_n, \sigma \rangle$, i.e., the transition function for input $\langle q_n, \sigma \rangle$ is undefined. In other words, the machine has no instruction to carry out, and at that point, it ceases operation. Halting is sometimes represented by a specific halt state h . This will be demonstrated in more detail later on.

The beauty of Turing's paper, "On computable numbers," is that he presents not only a formal definition, but also an argument that the definition captures the intuitive notion of computability. From the definition, it should be clear that any function computable by a Turing machine is computable in the intuitive sense. Turing offers three types of argument that the converse is true, i.e., that any function that we would naturally regard as computable is computable by such a machine. They are (in Turing's words):

1. A direct appeal to intuition.
2. A proof of the equivalence of two definitions (in case the new definition has a greater intuitive appeal).
3. Giving examples of large classes of numbers which are computable.

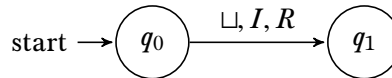
Our goal is to try to define the notion of computability "in principle," i.e., without taking into account practical limitations of time and space. Of course, with the broadest definition of computability in place, one can then go on to consider computation with bounded resources; this forms the heart of the subject known as "computational complexity."

Historical Remarks Alan Turing invented Turing machines in 1936. While his interest at the time was the decidability of first-order logic, the paper has been described as a definitive paper

on the foundations of computer design. In the paper, Turing focuses on computable real numbers, i.e., real numbers whose decimal expansions are computable; but he notes that it is not hard to adapt his notions to computable functions on the natural numbers, and so on. Notice that this was a full five years before the first working general purpose computer was built in 1941 (by the German Konrad Zuse in his parents living room), seven years before Turing and his colleagues at Bletchley Park built the code-breaking Colossus (1943), nine years before the American ENIAC (1945), twelve years before the first British general purpose computer the Manchester Mark I was built in Manchester (1948) and thirteen years before the Americans first tested the BINAC (1949). The Manchester Mark I has the distinction of being the first stored-program computer—previous machines had to be rewired by hand for each new task.

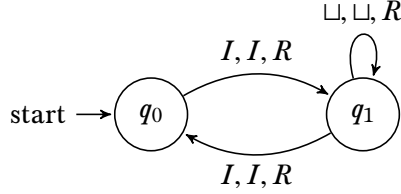
10.2 Representing Turing Machines

Turing machines can be represented visually by *state diagrams*. The diagrams are composed of state cells connected by arrows. Unsurprisingly, each state cell represents a state of the machine. Each arrow represents an instruction that can be carried out from that state, with the specifics of the instruction written above or below the appropriate arrow. Consider the following machine, which has only two internal states, q_0 and q_1 , and one instruction:



Recall that the Turing machine has a read/write head and a tape with the input written on it. The instruction can be read as *if reading a blank in state q_0 , write a stroke, move right, and move to state q_1* . This is equivalent to the transition function mapping $\langle q_0, \sqcup \rangle$ to $\langle q_1, I, R \rangle$.

Example 10.1. *Even Machine:* The following Turing machine halts if, and only if, there are an even number of strokes on the tape.



The state diagram corresponds to the following transition function:

$$\delta(q_0, I) = \langle q_1, I, R \rangle,$$

$$\delta(q_1, I) = \langle q_0, I, R \rangle,$$

$$\delta(q_0, \sqcup) = \langle q_0, \sqcup, R \rangle$$

The above machine halts only when the input is an even number of strokes. Otherwise, the machine (theoretically) continues to operate indefinitely. For any machine and input, it is possible to trace through the *configurations* of the machine in order to determine the output. We will give a formal definition of configurations later. For now, we can intuitively think of configurations as a series of diagrams showing the state of the machine at any point in time during operation. Configurations show the content of the tape, the state of the machine and the location of the read/write head.

Let us trace through the configurations of the even machine if it is started with an input of 4 *I*s. In this case, we expect that the machine will halt. We will then run the machine on an input of 3 *I*s, where the machine will run forever.

The machine starts in state q_0 , scanning the leftmost *I*. We can represent the initial state of the machine as follows:

$$\triangleright I_0 I I I \sqcup \dots$$

The above configuration is straightforward. As can be seen, the machine starts in state one, scanning the leftmost *I*. This is rep-

resented by a subscript of the state name on the first I . The applicable instruction at this point is $\delta(q_0, I) = \langle q_1, I, R \rangle$, and so the machine moves right on the tape and changes to state q_1 .

$$\triangleright I I_1 I I \sqcup \dots$$

Since the machine is now in state q_1 scanning a stroke, we have to “follow” the instruction $\delta(q_1, I) = \langle q_0, I, R \rangle$. This results in the configuration

$$\triangleright I I I_0 I \sqcup \dots$$

As the machine continues, the rules are applied again in the same order, resulting in the following two configurations:

$$\triangleright I I I I_1 \sqcup \dots$$

$$\triangleright I I I I \sqcup_0 \dots$$

The machine is now in state q_0 scanning a blank. Based on the transition diagram, we can easily see that there is no instruction to be carried out, and thus the machine has halted. This means that the input has been accepted.

Suppose next we start the machine with an input of three strokes. The first few configurations are similar, as the same instructions are carried out, with only a small difference of the tape input:

$$\triangleright I_0 I I \sqcup \dots$$

$$\triangleright I I_1 I \sqcup \dots$$

$$\triangleright I I I_0 \sqcup \dots$$

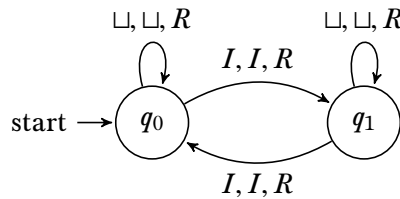
$$\triangleright I I I \sqcup_1 \dots$$

The machine has now traversed past all the strokes, and is reading a blank in state q_1 . As shown in the diagram, there is an instruction of the form $\delta(q_1, \sqcup) = \langle q_1, \sqcup, R \rangle$. Since the tape is infinitely blank to the right, the machine will continue to execute this instruction *forever*, staying in state q_1 and moving ever further

to the right. The machine will never halt, and does not accept the input.

It is important to note that not all machines will halt. If halting means that the machine runs out of instructions to execute, then we can create a machine that never halts simply by ensuring that there is an outgoing arrow for each symbol at each state. The even machine can be modified to run infinitely by adding an instruction for scanning a blank at q_0 .

Example 10.2.



Machine tables are another way of representing Turing machines. Machine tables have the tape alphabet displayed on the x -axis, and the set of machine states across the y -axis. Inside the table, at the intersection of each state and symbol, is written the rest of the instruction—the new state, new symbol, and direction of movement. Machine tables make it easy to determine in what state, and for what symbol, the machine halts. Whenever there is a gap in the table is a possible point for the machine to halt. Unlike state diagrams and instruction sets, where the points at which the machine halts are not always immediately obvious, any halting points are quickly identified by finding the gaps in the machine table.

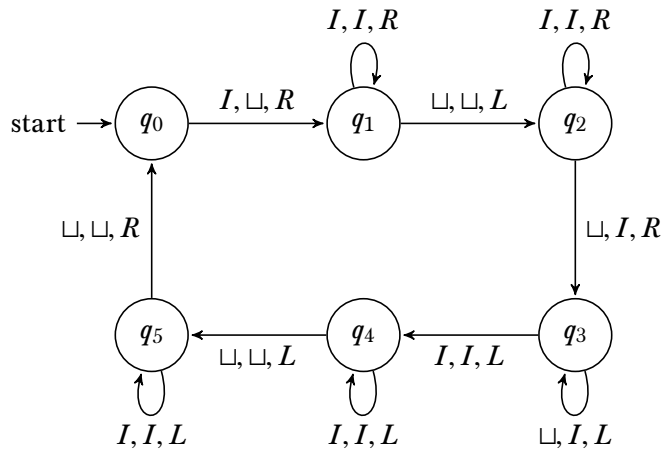
Example 10.3. The machine table for the even machine is:

	□	I
q_0		I, q_1, R
q_1	□, q_1 , □	I, q_0, R

As we can see, the machine halts when scanning a blank in state q_0 .

So far we have only considered machines that read and accept input. However, Turing machines have the capacity to both read and write. An example of such a machine (although there are many, many examples) is a *doubler*. A doubler, when started with a block of n strokes on the tape, outputs a block of $2n$ strokes.

Example 10.4. Before building a doubler machine, it is important to come up with a *strategy* for solving the problem. Since the machine (as we have formulated it) cannot remember how many strokes it has read, we need to come up with a way to keep track of all the strokes on the tape. One such way is to separate the output from the input with a blank. The machine can then erase the first stroke from the input, traverse over the rest of the input, leave a blank, and write two new strokes. The machine will then go back and find the second stroke in the input, and double that one as well. For each one stroke of input, it will write two strokes of output. By erasing the input as the machine goes, we can guarantee that no stroke is missed or doubled twice. When the entire input is erased, there will be $2n$ strokes left on the tape.



10.3 Turing Machines

The formal definition of what constitutes a Turing machine looks abstract, but is actually simple: it merely packs into one mathematical structure all the information needed to specify the workings of a Turing machine. This includes (1) which states the machine can be in, (2) which symbols are allowed to be on the tape, (3) which state the machine should start in, and (4) what the instruction set of the machine is.

Definition 10.5. A Turing machine $T = \langle Q, \Sigma, q_0, \delta \rangle$ consists of

1. a finite set of *states* Q ,
2. a finite *alphabet* Σ which includes \triangleright and \sqcup ,
3. an *initial state* $q_0 \in Q$,
4. a finite *instruction set* $\delta: Q \times \Sigma \rightarrow Q \times \Sigma \times \{L, R, N\}$.

The function δ is also called the *transition function* of T .

We assume that the tape is infinite in one direction only. For this reason it is useful to designate a special symbol \triangleright as a marker for the left end of the tape. This makes it easier for Turing machine programs to tell when they're "in danger" of running off the tape.

Example 10.6. Even Machine: The even machine is formally the quadruple $\langle Q, \Sigma, q_0, \delta \rangle$ where

$$\begin{aligned} Q &= \{q_0, q_1\} \\ \Sigma &= \{\triangleright, \sqcup, I\}, \\ \delta(q_0, I) &= \langle q_1, I, R \rangle, \\ \delta(q_1, I) &= \langle q_0, I, R \rangle, \\ \delta(q_0, \sqcup) &= \langle q_0, \sqcup, R \rangle. \end{aligned}$$

10.4 Configurations and Computations

Recall tracing through the configurations of the even machine earlier. The imaginary mechanism consisting of tape, read/write head, and Turing machine program is really just in intuitive way of visualizing what a Turing machine computation is. Formally, we can define the computation of a Turing machine on a given input as a sequence of *configurations*—and a configuration in turn is a sequence of symbols (corresponding to the contents of the tape at a given point in the computation), a number indicating the position of the read/write head, and a state. Using these, we can define what the Turing machine M computes on a given input.

Definition 10.7. A *configuration* of Turing machine $M = \langle Q, \Sigma, q_0, \delta \rangle$ is a triple $\langle C, n, q \rangle$ where

1. $C \in \Sigma^*$ is a finite sequence of symbols from Σ ,
2. $n \in \mathbb{N}$ is a number $< \text{len}(C)$, and
3. $q \in Q$

Intuitively, the sequence C is the content of the tape (symbols of all squares from the leftmost square to the last non-blank or previously visited square), n is the number of the square the read/write head is scanning (beginning with 0 being the number of the leftmost square), and q is the current state of the machine.

The potential input for a Turing machine is a sequence of symbols, usually a sequence that encodes a number in some form. The initial configuration of the Turing machine is that configuration in which we start the Turing machine to work on that input: the tape contains the tape end marker immediately followed by the input written on the squares to the right, the read/write head is scanning the leftmost square of the input (i.e., the square to the right of the left end marker), and the mechanism is in the designated start state q_0 .

Definition 10.8. The *initial configuration* of M for input $I \in \Sigma^*$ is

$$\langle \triangleright \frown I, 1, q_0 \rangle$$

The \frown symbol is for *concatenation*—we want to ensure that there are no blanks between the left end marker and the beginning of the input.

Definition 10.9. We say that a configuration $\langle C, n, q \rangle$ *yields* $\langle C', n', q' \rangle$ *in one step* (according to M), iff

1. the n -th symbol of C is σ ,
2. the instruction set of M specifies $\delta(q, \sigma) = \langle q', \sigma', D \rangle$,
3. the n -th symbol of C' is σ' , and
4.
 - a) $D = L$ and $n' = n - 1$, or
 - b) $D = R$ and $n' = n + 1$, or
 - c) $D = N$ and $n' = n$,
5. if $n' > \text{len}(C)$, then $\text{len}(C') = \text{len}(C) + 1$ and the n' -th symbol of C' is \sqcup .
6. for all i such that $i < \text{len}(C')$ and $i \neq n$, $C'(i) = C(i)$,

Definition 10.10. A *run of M on input I* is a sequence C_i of configurations of M , where C_0 is the initial configuration of M for input I , and each C_i yields C_{i+1} in one step.

We say that M *halts on input I after k steps* if $C_k = \langle C, n, q \rangle$, the n th symbol of C is σ , and $\delta(q, \sigma)$ is undefined. In that case, the *output* of M for input I is O , where O is a string of symbols not beginning or ending in \sqcup such that $C = \triangleright \frown \sqcup^i \frown O \frown \sqcup^j$ for some $i, j \in \mathbb{N}$.

According to this definition, the output O of M always begins and ends in a symbol other than \sqcup , or, if at time k the entire tape is filled with \sqcup (except for the leftmost \triangleright), O is the empty string.

10.5 Unary Representation of Numbers

Turing machines work on sequences of symbols written on their tape. Depending on the alphabet a Turing machine uses, these sequences of symbols can represent various inputs and outputs. Of particular interest, of course, are Turing machines which compute *arithmetical* functions, i.e., functions of natural numbers. A simple way to represent positive integers is by coding them as sequences of a single symbol I . If $n \in \mathbb{N}$, let I^n be the empty sequence if $n = 0$, and otherwise the sequence consisting of exactly n I 's.

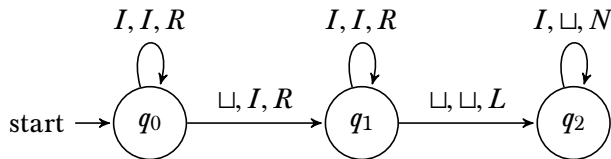
Definition 10.11. A Turing machine M computes the function $f: \mathbb{N}^n \rightarrow \mathbb{N}$ iff M halts on input

$$I^{k_1} \sqcup I^{k_2} \sqcup \dots \sqcup I^{k_n}$$

with output $I^{f(k_1, \dots, k_n)}$.

Example 10.12. Addition: Build a machine that, when given an input of two non-empty strings of I 's of length n and m , computes the function $f(n, m) = n + m$.

We want to come up with a machine that starts with two blocks of strokes on the tape and halts with one block of strokes. We first need a method to carry out. The input strokes are separated by a blank, so one method would be to write a stroke on the square containing the blank, and erase the first (or last) stroke. This would result in a block of $n + m$ I 's. Alternatively, we could proceed in a similar way to the doubler machine, by erasing a stroke from the first block, and adding one to the second block of strokes until the first block has been removed completely. We will proceed with the former example.

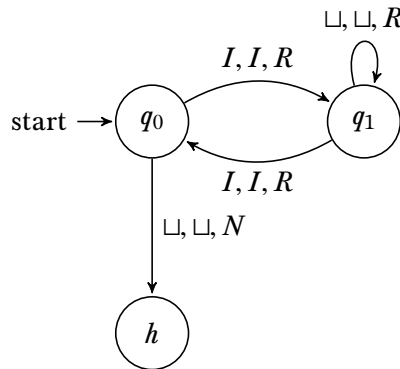


10.6 Halting States

Although we have defined our machines to halt only when there is no instruction to carry out, common representations of Turing machines have a dedicated *halting state*, h , such that $h \in Q$.

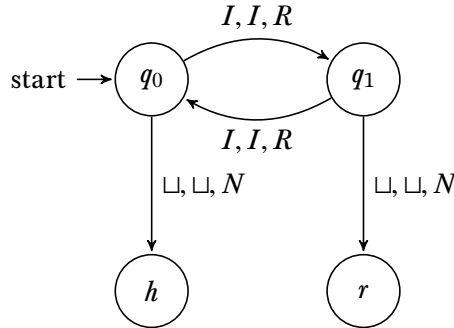
The idea behind a halting state is simple: when the machine has finished operation (it is ready to accept input, or has finished writing the output), it goes into a state h where it halts. Some machines have two halting states, one that accepts input and one that rejects input.

Example 10.13. Halting States. To elucidate this concept, let us begin with an alteration of the even machine. Instead of having the machine halt in state q_0 if the input is even, we can add an instruction to send the machine into a halt state.



Let us further expand the example. When the machine determines that the input is odd, it never halts. We can alter the machine to include a *reject* state by replacing the looping instruc-

tion with an instruction to go to a reject state r .



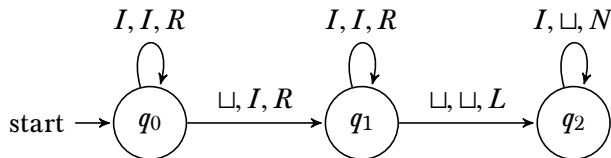
Adding a dedicated halting state can be advantageous in cases like this, where it makes explicit when the machine accepts/rejects certain inputs. However, it is important to note that no computing power is gained by adding a dedicated halting state. Similarly, a less formal notion of halting has its own advantages. The definition of halting used so far in this chapter makes the proof of the *Halting Problem* intuitive and easy to demonstrate. For this reason, we continue with our original definition.

10.7 Combining Turing Machines

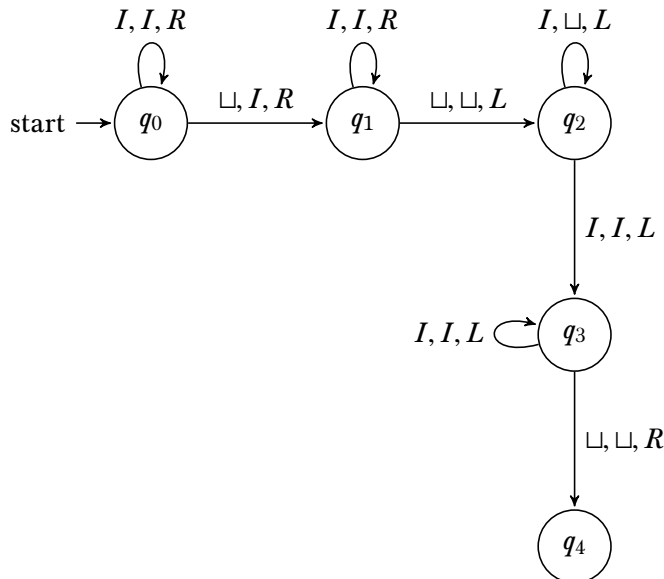
The examples of Turing machines we have seen so far have been fairly simple in nature. But in fact, any problem that can be solved with any modern programming language can also be solved with Turing machines. To build more complex Turing machines, it is important to convince ourselves that we can combine them, so we can build machines to solve more complex problems by breaking the procedure into simpler parts. If we can find a natural way to break a complex problem down into constituent parts, we can tackle the problem in several stages, creating several simple Turing machines and combining them into one machine that can solve the problem. This point is especially important when tackling the Halting Problem in the next section.

Example 10.14. Combining Machines: Design a machine that computes the function $f(m, n) = 2(m + n)$.

In order to build this machine, we can combine two machines we are already familiar with: the addition machine, and the doubler. We begin by drawing a state diagram for the addition machine.

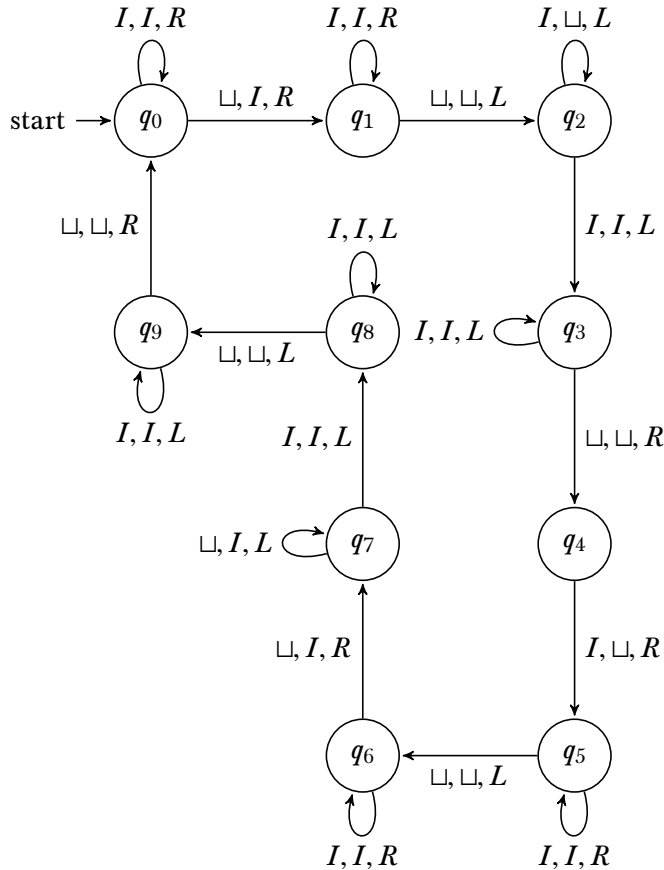


Instead of halting at state q_2 , we want to continue operation in order to double the output. Recall that the doubler machine erases the first stroke in the input and writes two strokes in a separate output. Let's add an instruction to make sure the tape head is reading the first stroke of the output of the addition machine.



It is now easy to double the input—all we have to do is connect the doubler machine onto state q_4 . This requires renaming the

states of the doubler machine so that they start at q_4 instead of q_0 —this way we don't end up with two starting states. The final diagram should look like:



10.8 Variants of Turing Machines

There are in fact many possible ways to define Turing machines, of which ours is only one. We allow arbitrary finite alphabets, a more restricted definition might allow only two tape symbols, I and \sqcup . We allow the machine to write a symbol to the tape

and move at the same time, other definitions allow either writing or moving. We allow the possibility of writing without moving the tape head, other definitions leave out the N “instruction.” Our definition assumes that the tape is infinite in one direction only, other definitions allow the tape to be infinite both to the left and the right. In fact, we might even allow any number of separate tapes, or even an infinite grid of squares. We represent the instruction set of the Turing machine by a transition function; other definitions use a transition relation.

This last relaxation of the definition is particularly interesting. In our definition, when the machine is in state q reading symbol σ , $\delta(q, \sigma)$ determines what the new symbol, state, and tape head position is. But if we allow the instruction set to be a relation between current state-symbol pairs $\langle q, \sigma \rangle$ and new state-symbol-direction triples $\langle q', \sigma', D \rangle$, the action of the Turing machine may not be uniquely determined—the instruction relation may contain both $\langle q, \sigma, q', \sigma', D \rangle$ and $\langle q, \sigma, q'', \sigma'', D' \rangle$. In this case we have a *non-deterministic* Turing machine. These play an important role in computational complexity theory.

There are also different conventions for when a Turing machine halts: we say it halts when the transition function is undefined, other definitions require the machine to be in a special designated halting state. And there are different ways of representing numbers: we use unary representation, but you can also use binary representation (this requires two symbols in addition to \sqcup).

Now here is an interesting fact: none of these variations matters as to which functions are Turing computable. If a function is Turing computable according to one definition, it is Turing computable according to all of them.

10.9 The Church-Turing Thesis

Turing machines are supposed to be a precise replacement for the concept of an effective procedure. Turing took it that anyone

who grasped the concept of an effective procedure and the concept of a Turing machine would have the intuition that anything that could be done via an effective procedure could be done by Turing machine. This claim is given support by the fact that all the other proposed precise replacements for the concept of an effective procedure turn out to be extensionally equivalent to the concept of a Turing machine—that is, they can compute exactly the same set of functions. This claim is called the *Church-Turing thesis*.

Definition 10.15. *The Church-Turing Thesis:* Anything computable via an effective procedure is Turing computable.

The Church-Turing thesis is appealed to in two ways. The first kind of use of the Church-Turing thesis is an excuse for laziness. Suppose we have a description of an effective procedure to compute something, say, in “pseudo-code.” Then we can invoke the Church-Turing thesis to justify the claim that the same function is computed by some Turing machine, even if we have not in fact constructed it.

The other use of the Church-Turing thesis is more philosophically interesting. It can be shown that there are functions which cannot be computed by a Turing machines. From this, using the Church-Turing thesis, one can conclude that it cannot be effectively computed, using any procedure whatsoever. For if there were such a procedure, by the Church-Turing thesis, it would follow that there would be a Turing machine. So if we can prove that there is no Turing machine that computes it, there also can’t be an effective procedure. In particular, the Church-Turing thesis is invoked to claim that the so-called halting problem not only cannot be solved by Turing machines, it cannot be effectively solved at all.

CHAPTER 11

Undecidability

11.1 Introduction

It might seem obvious that not every function, even every arithmetical function, can be computable. There are just too many, whose behavior is too complicated. Functions defined from the decay of radioactive particles, for instance, or other chaotic or random behavior. Suppose we start counting 1-second intervals from a given time, and define the function $f(n)$ as the number of particles in the universe that decay in the n -th 1-second interval after that initial moment. This seems like a candidate for a function we cannot ever hope to compute.

But it is one thing to not be able to imagine how one would compute such functions, and quite another to actually prove that they are uncomputable. In fact, even functions that seem hopelessly complicated may, in an abstract sense, be computable. For instance, suppose the universe is finite in time—some day, in the very distant future the universe will contract into a single point, as some cosmological theories predict. Then there is only a finite (but incredibly large) number of seconds from that initial moment for which $f(n)$ is defined. And any function which is defined for only finitely many inputs is computable: we could list the outputs in one big table, or code it in one very big Turing machine state transition diagram.

We are often interested in special cases of functions whose values give the answers to yes/no questions. For instance, the question “is n a prime number?” is associated with the function

$$\text{isprime}(n) = \begin{cases} 1 & \text{if } n \text{ is prime} \\ 0 & \text{otherwise.} \end{cases}$$

We say that a yes/no question can be *effectively decided*, if the associated 1/0-valued function is effectively computable.

To prove mathematically that there are functions which cannot be effectively computed, or problems that cannot effectively decided, it is essential to fix a specific model of computation, and show about it that there are functions it cannot compute or problems it cannot decide. We can show, for instance, that not every function can be computed by Turing machines, and not every problem can be decided by Turing machines. We can then appeal to the Church-Turing thesis to conclude that not only are Turing machines not powerful enough to compute every function, but no effective procedure can.

The key to proving such negative results is the fact that we can assign numbers to Turing machines themselves. The easiest way to do this is to enumerate them, perhaps by fixing a specific way to write down Turing machines and their programs, and then listing them in a systematic fashion. Once we see that this can be done, then the existence of Turing-uncomputable functions follows by simple cardinality considerations: there functions from \mathbb{N} to \mathbb{N} (in fact, even just from \mathbb{N} to $\{0, 1\}$) are uncountable, but since we can enumerate all the Turing machines, the Turing-computable functions are only countably infinite.

We can also define *specific* functions and problems which we can prove to be uncomputable and undecidable, respectively. One such problem is the so-called *Halting Problem*. Turing machines can be finitely described by listing their instructions. Such a description of a Turing machine, i.e., a Turing machine program, can of course be used as input to another Turing machine. So we can consider Turing machines that decide questions about

other Turing machines. One particularly interesting question is this: “Does the given Turing machine eventually halt when started on input n ?” It would be nice if there were a Turing machine that could decide this question: think of it as a quality-control Turing machine which ensures that Turing machines don’t get caught in infinite loops and such. The interesting fact, which Turing proved, is that there cannot be such a Turing machine. There cannot be a single Turing machine which, when started on input consisting of a description of a Turing machine M and some number n , will always halt with either output 1 or 0 according to whether M machine would have halted when started on input n or not.

Once we have examples of specific undecidable problems we can use them to show that other problems are undecidable, too. For instance, one celebrated undecidable problem is the question, “Is the first-order formula A valid?”. There is no Turing machine which, given as input a first-order formula A , is guaranteed to halt with output 1 or 0 according to whether A is valid or not. Historically, the question of finding a procedure to effectively solve this problem was called simply “the” decision problem; and so we say that the decision problem is unsolvable. Turing and Church proved this result independently at around the same time, so it is also called the Church-Turing Theorem.

11.2 Enumerating Turing Machines

We can show that the set of all Turing-machines is countable. This follows from the fact that each Turing machine can be finitely described. The set of states and the tape vocabulary are finite sets. The transition function is a partial function from $Q \times \Sigma$ to $Q \times \Sigma \times \{L, R, N\}$, and so likewise can be specified by listing its values for the finitely many argument pairs for which it is defined. Of course, strictly speaking, the states and vocabulary can be anything; but the *behavior* of the Turing machine is independent of which objects serve as states and vocabulary. So we may

assume, for instance, that the states and vocabulary symbols are natural numbers, or that the states and vocabulary are all strings of letters and digits.

Suppose we fix a countably infinite vocabulary for specifying Turing machines: $\sigma_0 = \triangleright$, $\sigma_1 = \sqcup$, $\sigma_2 = I$, $\sigma_3, \dots, R, L, N, q_0, q_1, \dots$. Then any Turing machine can be specified by some finite string of symbols from this alphabet (though not every finite string of symbols specifies a Turing machine). For instance, suppose we have a Turing machine $M = \langle Q, \Sigma, q, \delta \rangle$ where

$$\begin{aligned} Q &= \{q'_0, \dots, q'_n\} \subseteq \{q_0, q_1, \dots\} \text{ and} \\ \Sigma &= \{\triangleright, \sigma'_1, \sigma'_2, \dots, \sigma'_m\} \subseteq \{\sigma_0, \sigma_1, \dots\}. \end{aligned}$$

We could specify it by the string

$$q'_0 q'_1 \dots q'_n \triangleright \sigma'_1 \dots \sigma'_m \triangleright q \triangleright S(\sigma'_0, q'_0) \triangleright \dots \triangleright S(\sigma'_m, q'_n)$$

where $S(\sigma'_i, q'_j)$ is the string $\sigma'_i q'_j \delta(\sigma'_i, q'_j)$ if $\delta(\sigma'_i, q'_j)$ is defined, and $\sigma'_i q'_j$ otherwise.

Theorem 11.1. *There are functions from \mathbb{N} to \mathbb{N} which are not Turing computable.*

Proof. We know that the set of finite strings of symbols from a countably infinite alphabet is countable. This gives us that the set of descriptions of Turing machines, as a subset of the finite strings from the countable vocabulary $\{q_0, q_1, \dots, \triangleright, \sigma_1, \sigma_2, \dots\}$, is itself enumerable. Since every Turing computable function is computed by some (in fact, many) Turing machines, this means that the set of all Turing computable functions from \mathbb{N} to \mathbb{N} is also enumerable.

On the other hand, the set of all functions from \mathbb{N} to \mathbb{N} is not countable. This follows immediately from the fact that not even the set of all functions of one argument from \mathbb{N} to the set $\{0, 1\}$ is countable. If all functions were computable by some Turing machine we could enumerate the set of all functions. So there are some functions that are not Turing-computable. \square

11.3 The Halting Problem

Assume we have fixed some finite descriptions of Turing machines. Using these, we can enumerate Turing machines via their descriptions, say, ordered by the lexicographic ordering. Each Turing machine thus receives an *index*: its place in the enumeration M_1, M_2, M_3, \dots of Turing machine descriptions.

We know that there must be non-Turing-computable functions: the set of Turing machine descriptions—and hence the set of Turing machines—is enumerable, but the set of all functions from \mathbb{N} to \mathbb{N} is not. But we can find specific examples of non-computable function as well. One such function is the halting function.

Definition 11.2. The *halting function* h is defined as

$$h(e, n) = \begin{cases} 0 & \text{if machine } M_e \text{ does not halt for input } n \\ 1 & \text{if machine } M_e \text{ halts for input } n \end{cases}$$

Definition 11.3. The *Halting Problem* is the problem of determining (for any m, w) whether the Turing machine M_e halts for an input of n strokes.

We show that h is not Turing-computable by showing that a related function, s , is not Turing-computable. This proof relies on the fact that anything that can be computed by a Turing machine can be computed using just two symbols: \sqcup and I , and the fact that two Turing machines can be hooked together to create a single machine.

Definition 11.4. The function s is defined as

$$s(e) = \begin{cases} 0 & \text{if machine } M_e \text{ does not halt for input } e \\ 1 & \text{if machine } M_e \text{ halts for input } e \end{cases}$$

Lemma 11.5. *The function s is not Turing computable.*

Proof. We suppose, for contradiction, that the function s is Turing-computable. Then there would be a Turing machine S that computes s . We may assume, without loss of generality, that when S halts, it does so while scanning the first square. This machine can be “hooked up” to another machine J , which halts if it is started on a blank tape (i.e., if it reads \sqcup in the initial state while scanning the square to the right of the end-of-tape symbol), and otherwise wanders off to the right, never halting. $S \frown J$, the machine created by hooking S to J , is a Turing machine, so it is M_e for some e (i.e., it appears somewhere in the enumeration). Start M_e on an input of e I s. There are two possibilities: either M_e halts or it does not halt.

1. Suppose M_e halts for an input of e I s. Then $s(e) = 1$. So S , when started on e , halts with a single I as output on the tape. Then J starts with a I on the tape. In that case J does not halt. But M_e is the machine $S \frown J$, so it should do exactly what S followed by J would do. So M_e cannot halt for an input of e I 's.
2. Now suppose M_e does not halt for an input of e I s. Then $s(e) = 0$, and S , when started on input e , halts with a blank tape. J , when started on a blank tape, immediately halts. Again, M_e does what S followed by J would do, so M_e must halt for an input of e I 's.

This shows there cannot be a Turing machine S : s is not Turing computable. \square

Theorem 11.6 (Unsolvability of the Halting Problem). *The halting problem is unsolvable, i.e., the function h is not Turing computable.*

Proof. Suppose h were Turing computable, say, by a Turing machine H . We could use H to build a Turing machine that computes s : First, make a copy of the input (separated by a blank). Then move back to the beginning, and run H . We can clearly

make a machine that does the former, and if H existed, we would be able to “hook it up” to such a modified doubling machine to get a new machine which would determine if M_e halts on input e , i.e., computes s . But we’ve already shown that no such machine can exist. Hence, h is also not Turing computable. \square

11.4 The Decision Problem

We say that first-order logic is *decidable* iff there is an effective method for determining whether or not a given sentence is valid. AS it turns out, there is no such method: the problem of deciding validity of first-order sentences is unsolvable.

In order to establish this important negative result, we prove that the decision problem cannot be solved by a Turing machine. That is, we show that there is no Turing machine which, whenever it is started on a tape that contains a first-order sentence, eventually halts and outputs either 1 or 0 depending on whether the sentence is valid or not. By the Church-Turing thesis, every function which is computable is Turing computable. So if this “validity function” were effectively computable at all, it would be Turing computable. If it isn’t Turing computable, then, it also cannot be effectively computable.

Our strategy for proving that the decision problem is unsolvable is to reduce the halting problem to it. This means the following: We have proved that the function $h(e, w)$ that halts with output 1 if the Turing-machine described by e halts on input w and outputs 0 otherwise, is not Turing-computable. We will show that if there were a Turing machine that decides validity of first-order sentences, then there is also Turing machine that computes h . Since h cannot be computed by a Turing machine, there cannot be a Turing machine that decides validity either.

The first step in this strategy is to show that for every input w and a Turing machine M , we can effectively describe a sentence T representing M and w and a sentence E expressing “ M eventually halts” such that:

$\models T \rightarrow E$ iff M halts for input w .

The bulk of our proof will consist in describing these sentences $T(M, w)$ and $E(M, w)$ and verifying that $T(M, w) \rightarrow E(M, w)$ is valid iff M halts on input w .

11.5 Representing Turing Machines

In order to represent Turing machines and their behavior by a sentence of first-order logic, we have to define a suitable language. The language consists of two parts: predicate symbols for describing configurations of the machine, and expressions for counting execution steps (“moments”) and positions on the tape. The latter require an initial moment, o , a “successor” function which is traditionally written as a postfix ι , and an ordering $x < y$ of “before.”

Definition 11.7. Given a Turing machine $M = \langle Q, \Sigma, q_0, \delta \rangle$, the language \mathcal{L}_M consists of:

1. A two-place predicate symbol $Q_q(x, y)$ for every state $q \in Q$. Intuitively, $Q_q(\bar{n}, \bar{m})$ expresses “after m steps, M is in state q scanning the n th square.”
2. A two-place predicate symbol $S_\sigma(x, y)$ for every symbol $\sigma \in \Sigma$. Intuitively, $S_\sigma(\bar{n}, \bar{m})$ expresses “after m steps, the n th square contains symbol σ .”
3. A constant o
4. A one-place function ι
5. A two-place predicate $<$

For each number n there is a canonical term \bar{n} , the *numeral* for n , which represents it in \mathcal{L}_M . $\bar{0}$ is o , $\bar{1}$ is o' , $\bar{2}$ is o'' , and so

on. More formally:

$$\begin{aligned}\bar{0} &= 0 \\ \overline{n+1} &= \bar{n}'\end{aligned}$$

The sentences describing the operation of the Turing machine M on input $w = \sigma_{i_1} \dots \sigma_{i_n}$ are the following:

1. Axioms describing numbers:

- a) A sentence that says that the successor function is injective:

$$\forall x \forall y (x' = y' \rightarrow x = y)$$

- b) A sentence that says that every number is less than its successor:

$$\forall x (x < x')$$

- c) A sentence that ensures that $<$ is transitive:

$$\forall x \forall y \forall z ((x < y \wedge y < z) \rightarrow x < z)$$

2. Axioms describing the input configuration:

- a) M is in the initial state q_0 at time 0, scanning square 1:

$$Q_{q_0}(\bar{1}, \bar{0})$$

- b) The first $n+1$ squares contain the symbols $\triangleright, \sigma_1, \dots, \sigma_n$:

$$S_{\triangleright}(\bar{0}, \bar{0}) \wedge S_{\sigma_1}(\bar{1}, \bar{0}) \wedge \dots \wedge S_{\sigma_n}(\bar{n}, \bar{0})$$

- c) Otherwise, the tape is empty:

$$\forall x (\bar{n} < x \rightarrow S_{\sqcup}(x, \bar{0}))$$

3. Axioms describing the transition from one configuration to the next:

For the following, let $A(x, y)$ be the conjunction of all sentences of the form

$$\forall z (((z < x \vee x < z) \wedge S_\sigma(z, y)) \rightarrow S_\sigma(z, y'))$$

where $\sigma \in \Sigma$. We use $A(\bar{n}, \bar{m})$ to express “other than at square n , the tape after $m + 1$ steps is the same as after m steps.”

- a) For every instruction $\delta(q_i, \sigma) = \langle q_j, \sigma', R \rangle$, the sentence:

$$\begin{aligned} \forall x \forall y ((Q_{q_i}(x, y) \wedge S_\sigma(x, y)) \rightarrow \\ (Q_{q_j}(x', y') \wedge S_{\sigma'}(x, y') \wedge A(x, y))) \end{aligned}$$

- b) For every instruction $\delta(q_i, \sigma) = \langle q_j, \sigma', L \rangle$, the sentence:

$$\begin{aligned} \forall x \forall y ((Q_{q_i}(x', y) \wedge S_\sigma(x', y)) \rightarrow \\ (Q_{q_j}(x, y') \wedge S_{\sigma'}(x', y') \wedge A(x, y))) \end{aligned}$$

Take a moment to think about how this works: now we don't start with “if scanning square $x \dots$ ” but: “if scanning square $x + 1 \dots$ ” A move to the left means that in the next step the machine is scanning square x . But the square that is written on is $x + 1$. We do it this way since we don't have subtraction or a predecessor function.

- c) For every instruction $\delta(q_i, \sigma) = \langle q_j, \sigma', N \rangle$, the sentence:

$$\begin{aligned} \forall x \forall y ((Q_{q_i}(x, y) \wedge S_\sigma(x, y)) \rightarrow \\ (Q_{q_j}(x, y') \wedge S_{\sigma'}(x, y') \wedge A(x, y))) \end{aligned}$$

Let $T(M, w)$ be the conjunction of all the above sentences for Turing machine M and input w

In order to express that M eventually halts, we have to find a sentence that says “after some number of steps, the transition function will be undefined.” Let X be the set of all pairs $\langle q, \sigma \rangle$ such that $\delta(q, \sigma)$ is undefined. Let $E(M, w)$ then be the sentence

$$\exists x \exists y \left(\bigvee_{\langle q, \sigma \rangle \in X} (Q_q(x, y) \wedge S_\sigma(x, y)) \right)$$

If we use a Turing machine with a designated halting state h , it is even easier: then the sentence $E(M, w)$

$$\exists x \exists y Q_h(x, y)$$

expresses that the machine eventually halts.

11.6 Verifying the Representation

In order to verify that our representation works, we first have to make sure that if M halts on input w , then $T(M, w) \rightarrow E(M, w)$ is valid. We can do this simply by proving that $T(M, w)$ implies a description of the configuration of M for each step of the execution of M on input w . If M halts on input w , then for some n , M will be in a halting configuration at step n (and be scanning square m , for some m). Hence, $T(M, w)$ implies $Q_q(\bar{m}, \bar{n}) \wedge S_\sigma(\bar{m}, \bar{n})$ for some q and σ such that $\delta(q, \sigma)$ is undefined.

Definition 11.8. Let $C(M, w, n)$ be the sentence

$$Q_q(\bar{m}, \bar{n}) \wedge S_{\sigma_0}(\bar{0}, \bar{n}) \wedge \cdots \wedge S_{\sigma_k}(\bar{k}, \bar{n}) \wedge \forall x (\bar{k} < x \rightarrow S_\sqcup(x, \bar{n}))$$

where q is the state of M at time n , M is scanning square m at time n , square i contains symbol σ_i at time n for $0 \leq i \leq k$ and k is the right-most non-blank square of the tape at time m .

Suppose that M does halt for input w . Then there is some time n , state q , square m , and symbol σ such that:

1. At time n the machine is in state q scanning square m on which σ appears.

2. There transition function $\delta(q, \sigma)$ is undefined.

$C(M, w, n)$ will be the description of this time and will include the clauses $Q_q(\overline{m}, \overline{n})$ and $S_\sigma(\overline{m}, \overline{n})$. These clauses together imply $E(M, w)$:

$$\exists x \exists y \left(\bigvee_{\langle q, \sigma \rangle \in X} (Q_q(x, y) \wedge S_\sigma(x, y)) \right)$$

since $Q_{q'}(\overline{m}, \overline{n}) \wedge S_{\sigma'}(\overline{m}, \overline{n}) \models \bigvee_{\langle q, \sigma \rangle \in X} (Q_q(\overline{m}, \overline{n}) \wedge S_\sigma(\overline{m}, \overline{n}))$, as $\langle q', \sigma' \rangle \in X$.

So if M halts for input w , then there is some time n such that $C(M, w, n) \models E(M, w)$

Since consequence is transitive, it is sufficient to show that for any time n , $T(M, w) \models C(M, w, n)$.

Lemma 11.9. *For each n , $T(M, w) \models C(M, w, n)$.*

Proof. If $n = 0$, then the conjuncts of $C(M, w, 0)$ are also conjuncts of $T(M, w)$, so entailed by it.

Inductive hypothesis: If M has not halted before time n , then $T(M, w) \models C(M, w, n)$.

Suppose $n > 0$ and at time n , M started on w is in state q scanning square m , and the content of the tape is $\sigma_0, \dots, \sigma_k$.

Suppose that M has not halted before time $n + 1$. If $T(M, n)$ is true in a structure M , the inductive hypothesis tells us that $C(M, w, n)$ is true in M also. In particular, $Q_q(\overline{m}, \overline{n})$ and $S_\sigma(\overline{m}, \overline{n})$ are true in M .

Since M does not halt at n , there must be an instruction of one of the following three forms in the program of M :

1. $\delta(q, \sigma) = \langle q', \sigma', R \rangle$
2. $\delta(q, \sigma) = \langle q', \sigma', L \rangle$
3. $\delta(q, \sigma) = \langle q', \sigma', N \rangle$

We will consider each of these three cases in turn. First, assume that $m \leq k$.

1. Suppose there is an instruction of the form (1). By **Definition 11.7, (3a)**, this means that

$$\forall x \forall y ((Q_q(x, y) \wedge S_\sigma(x, y)) \rightarrow (Q_{q'}(x', y') \wedge S_{\sigma'}(x, y') \wedge A(x, y)))$$

is a conjunct of $T(M, w)$. This entails the following sentence, through universal instantiation:

$$(Q_q(\bar{m}, \bar{n}) \wedge S_\sigma(\bar{m}, \bar{n})) \rightarrow (Q_{q'}(\bar{m}', \bar{n}') \wedge S_{\sigma'}(\bar{m}, \bar{n}') \wedge A(m, n)).$$

This in turn entails,

$$\begin{aligned} & Q_{q'}(\bar{m}', \bar{n}') \wedge S_{\sigma'}(\bar{m}, \bar{n}') \wedge \\ & S_{\sigma_0}(\bar{0}, \bar{n}') \wedge \cdots \wedge S_{\sigma_k}(\bar{k}, \bar{n}') \wedge \\ & \forall x (\bar{k} < x \rightarrow S_\sqcup(x, \bar{n}')) \end{aligned}$$

The first line comes directly from the consequent of the preceding conditional. The each conjunct in the middle line—which excludes $S_{\sigma_m}(\bar{m}, \bar{n}')$ —follows from the corresponding conjunct in $C(M, w, n)$ together with $A(m, n)$. The last line follows from the corresponding conjunct in $C(M, w, n)$, $\bar{m} < x \rightarrow \bar{k} < x$, and $A(m, n)$. Together, this just is $C(M, w, n')$.

2. Suppose there is an instruction of the form (2). Then, by **Definition 11.7, (3b)**,

$$\forall x \forall y ((Q_q(x', y) \wedge S_\sigma(x', y)) \rightarrow (Q_{q'}(x, y') \wedge S_{\sigma'}(x', y') \wedge A(x, y)))$$

is a conjunct of $T(M, w)$, which entails the following sentence:

$$(Q_q(\bar{m}', \bar{n}) \wedge S_\sigma(\bar{m}', \bar{n})) \rightarrow (Q_{q'}(\bar{m}, \bar{n}') \wedge S_{\sigma'}(\bar{m}', \bar{n}') \wedge A(x, y)),$$

which in turn implies

$$\begin{aligned} Q_{q'}(\bar{m}, \bar{n}') \wedge S_{\sigma'_m}(\bar{m}, \bar{n}') \wedge \\ S_{\sigma_0}(\bar{0}, \bar{n}') \cdots \wedge S_{\sigma_k}(\bar{k}, \bar{n}') \wedge \\ \forall x (\bar{k} < x \rightarrow S_{\sqcup}(x, \bar{n}')) \end{aligned}$$

as before. But this just is $C(M, w, n')$.

3. Case (3) is left as an exercise.

If $m > k$ and $\sigma' \neq \sqcup$, the last instruction has written a non-blank symbol to the right of the right-most non-blank square k at time n . In this case, $C(M, w, n')$ has the form

$$\begin{aligned} Q_{q'}(\bar{m}', \bar{n}') \wedge \\ S_{\sigma_0}(\bar{0}, \bar{n}') \wedge \cdots \wedge S_{\sigma_k}(\bar{k}, \bar{n}') \wedge \\ S_{\sqcup}(\bar{k} + 1, \bar{n}') \wedge \cdots \wedge S_{\sqcup}(\bar{m} - 1, \bar{n}') \wedge \\ S_{\sigma'}(\bar{m}, \bar{n}') \wedge \\ \forall x (\bar{m} < x \rightarrow S_{\sqcup}(x, \bar{n}')) \end{aligned}$$

For $k < i < m$, $S_{\sqcup}(\bar{i}, \bar{n})$ follows from the conjunct $\forall x (\bar{k} < x \rightarrow S_{\sqcup}(x, \bar{n}))$ of $C(M, w, n)$ and the fact that $T(M, w) \models \bar{k} < \bar{i}$ if $k < i$. $S_{\sqcup}(\bar{i}, \bar{n}')$ then follows from $A(m, n)$ and $\bar{i} < \bar{m}$. From $\forall x (\bar{k} < x \rightarrow S_{\sqcup}(x, \bar{n}))$ we get $\forall x (\bar{m} < x \rightarrow S_{\sqcup}(x, \bar{n}))$ since $\bar{k} < \bar{m}$ and $<$ is transitive. From that plus $A(m, n)$ we get $\forall x (\bar{m} < x \rightarrow S_{\sqcup}(x, \bar{n}'))$. Similarly for cases (2) and (3).

We have shown that for any n , $T(M, w) \models C(M, w, n)$. \square

Lemma 11.10. *If M halts on input w , then $T(M, w) \rightarrow E(M, w)$ is valid.*

Proof. By Lemma 11.9, we know that, for any time n , the description $C(M, w, n)$ of the configuration of M at time n is a consequence of $T(M, w)$. Suppose M halts after k steps. It will

be scanning square m , say. Then $C(M, w, k)$ contains as conjuncts both $Q_q(\bar{m}, \bar{k})$ and $S_\sigma(\bar{m}, \bar{k})$ with $\delta(q, \sigma)$ undefined. Thus, $C(M, w, k) \models E(M, w)$. But then $T(M, w) \models E(M, w)$ and therefore $T(M, w) \rightarrow E(M, w)$ is valid. \square

To complete the verification of our claim, we also have to establish the reverse direction: if $T(M, w) \rightarrow E(M, w)$ is valid, then M does in fact halt when started on input w .

Lemma 11.11. *If $\models T(M, w) \rightarrow E(M, w)$, then M halts on input w .*

Proof. Consider the \mathcal{L}_M -structure M with domain \mathbb{N} which interprets 0 as 0, ' as the successor function, and $<$ as the less-than relation, and the predicates Q_q and S_σ as follows:

$$\begin{aligned} Q_q^M &= \{ \langle m, n \rangle : \begin{array}{l} \text{started on } w, \text{ after } n \text{ steps,} \\ M \text{ is in state } q \text{ scanning square } m \end{array} \} \\ S_\sigma^M &= \{ \langle m, n \rangle : \begin{array}{l} \text{started on } w, \text{ after } n \text{ steps,} \\ \text{square } m \text{ of } M \text{ contains symbol } \sigma \end{array} \} \end{aligned}$$

In other words, we construct the structure M so that it describes what M started on input w actually does, step by step. Clearly, $M \models T(M, w)$. If $\models T(M, w) \rightarrow E(M, w)$, then also $M \models E(M, w)$, i.e.,

$$M \models \exists x \exists y \left(\bigvee_{\langle q, \sigma \rangle \in X} (Q_q(x, y) \wedge S_\sigma(x, y)) \right).$$

As $|M| = \mathbb{N}$, there must be $m, n \in \mathbb{N}$ so that $M \models Q_q(\bar{m}, \bar{n}) \wedge S_\sigma(\bar{m}, \bar{n})$ for some q and σ such that $\delta(q, \sigma)$ is undefined. By the definition of M , this means that M started on input w after n steps is in state q and reading symbol σ , and the transition function is undefined, i.e., M has halted. \square

11.7 The Decision Problem is Unsolvable

Theorem 11.12. *The decision problem is unsolvable.*

Proof. Suppose the decision problem were solvable, i.e., suppose there were a Turing machine D of the following sort. Whenever D is started on a tape that contains a sentence B of first-order logic as input, D eventually halts, and outputs 1 iff B is valid and 0 otherwise. Then we could solve the halting problem as follows. We construct a Turing machine E that, given as input the number e of Turing machine M_e and input w , computes the corresponding sentence $T(M_e, w) \rightarrow E(M_e, w)$ and halts, scanning the leftmost square on the tape. The machine $E \frown D$ would then, given input e and w , first compute $T(M_e, w) \rightarrow E(M_e, w)$ and then run the decision problem machine D on that input. D halts with output 1 iff $T(M_e, w) \rightarrow E(M_e, w)$ is valid and outputs 0 otherwise. By Lemma 11.11 and Lemma 11.10, $T(M_e, w) \rightarrow E(M_e, w)$ is valid iff M_e halts on input w . Thus, $E \frown D$, given input e and w halts with output 1 iff M_e halts on input w and halts with output 0 otherwise. In other words, $E \frown D$ would solve the halting problem. But we know, by Theorem 11.6, that no such Turing machine can exist. \square



Emmy Noether

1882 - 1935

APPENDIX A

Induction

Induction is an important proof technique which is used, in different forms, in almost all areas of logic, theoretical computer science, and mathematics. It is needed to prove many of the results in this book.

A.1 Induction on \mathbb{N}

In its simplest form, induction is a technique used to prove results for all natural numbers. It uses the fact that by starting from 0 and repeatedly adding 1 we eventually reach every natural number. So to prove that something is true for every number, we can (1) establish that it is true for 0 and (2) show that whenever a number has it, the next number has it too. If we abbreviate “number n has property P ” by $P(n)$, then a proof by induction that $P(n)$ for all $n \in \mathbb{N}$ consists of:

1. a proof of $P(0)$, and
2. a proof that, for any n , if $P(n)$ then $P(n + 1)$.

To make this crystal clear, suppose we have both (1) and (2). Then (1) tells us that $P(0)$ is true. If we also have (2), we know in particular that if $P(0)$ then $P(0 + 1)$, i.e., $P(1)$. (This follows from the general statement “for any n , if $P(n)$ then $P(n + 1)$ ” by

putting 0 for n . So by modus ponens, we have that $P(1)$. From (2) again, now taking 1 for n , we have: if $P(1)$ then $P(2)$. Since we've just established $P(1)$, by modus ponens, we have $P(2)$. And so on. For any number k , after doing this k steps, we eventually arrive at $P(k)$. So (1) and (2) together established $P(k)$ for any $k \in \mathbb{N}$.

Let's look at an example. Suppose we want to find out how many different sums we can throw with n dice. Although it might seem silly, let's start with 0 dice. If you have no dice there's only one possible sum you can "throw": no dots at all, which sums to 0. So the number of different possible throws is 1. If you have only one die, i.e., $n = 1$, there are six possible values, 1 through 6. With two dice, we can throw any sum from 2 through 12, that's 11 possibilities. With three dice, we can throw any number from 3 to 18, i.e., 16 different possibilities. 1, 6, 11, 16: looks like a pattern: maybe the answer is $5n + 1$? Of course, $5n + 1$ is the maximum possible, because there are only $5n + 1$ numbers between n , the lowest value you can throw with n dice (all 1's) and $6n$, the highest you can throw (all 6's).

Theorem A.1. *With n dice one can throw all $5n + 1$ possible values between n and $6n$.*

Proof. To prove that this holds for any number of dice, we use induction: we prove, (1) that the number of different throws with 0 dice is $5 \cdot 0 + 1 = 1$, and (2) that if we can throw all $5n + 1$ values between n and $6n$ with n dice, then with $n + 1$ dice we can throw all $5(n + 1) + 1 = 5n + 6$ possible values between $n + 1$ and $6(n + 1)$. We've already established (1): If you have no dice, there's only one possible outcome: you throw all 0 dice, and the number of dots visible is 0.

Now let's see how many possible total values you can throw with $n + 1$ dice: we're going to show that it's all $5n + 6$ different sums between $n + 1$ and $6n + 6$, assuming that you can throw all $5n + 1$ different sums between n and $6n$ with n dice. This

assumption is called the induction hypothesis. So: assume that you can throw all $5n + 1$ different sums between n and $6n$ with n dice

Suppose you have n dice. Each possible throw sums to some number between n (all n dice show 1's) and $6n$ (all 6's). Now throw another die. You get another number between 1 and 6. If your total is less than $6n$ you could have thrown the same total with your original n dice, by induction hypothesis. Only the possible totals $6n+1$ through $6n+6$ are totals you couldn't have thrown with just n dice—and these are new possible totals, which you'd get, e.g., by throwing n 6's plus whatever your new die shows. So there are 6 new possible totals. You can still throw any of the old totals, except one. With $n + 1$ dice you can't roll just n . But any other value you can throw with n dice you can also throw with $n + 1$ dice: just imagine one of your n dice showed one eye fewer (and at least one of them must show at least a 2) and your $(n + 1)$ st die shows a 1. With an additional die we lose only one possible total (n) and gain 6 new possible totals ($6n + 1$ through $6n + 6$). So the number of possible totals with $n + 1$ dice is $5n + 1$, minus 1, plus 6, i.e., $5n + 6$. So we have proved that, assuming the number of totals we can throw with n dice is $5n + 1$, the number of totals we can throw with $n + 1$ dice is $5n + 6 = 5(n + 1) + 1$. \square

Very often we use induction when we want to prove something about a series of objects (numbers, sets, etc.) that is itself defined “inductively,” i.e., by defining the $(n+1)$ -st object in terms of the n -th. For instance, we can define the sum s_n of the natural numbers up to n by

$$\begin{aligned}s_0 &= 0 \\ s_{n+1} &= s_n + (n + 1)\end{aligned}$$

This definition gives:

$$\begin{aligned}
 s_0 &= 0, \\
 s_1 &= s_0 + 1 &= 1, \\
 s_2 &= s_1 + 2 &= 1 + 2 = 3 \\
 s_3 &= s_2 + 3 &= 1 + 2 + 3 = 6, \text{ etc.}
 \end{aligned}$$

Now we can prove, by induction, that $s_n = n(n+1)/2$.

Proposition A.2. $s_n = n(n+1)/2$.

Proof. We have to prove (1) that $s_0 = 0 \cdot (0+1)/2$ and (2) if $s_n = n(n+1)/2$ then $s_{n+1} = (n+1)(n+2)/2$. (1) is obvious. To prove (2), we assume the inductive hypothesis: $s_n = n(n+1)/2$. Using it, we have to show that $s_{n+1} = (n+1)(n+2)/2$.

What is s_{n+1} ? By the definition, $s_{n+1} = s_n + (n+1)$. By inductive hypothesis, $s_n = n(n+1)/2$. We can substitute this into the previous equation, and then just need a bit of arithmetic of fractions:

$$\begin{aligned}
 s_{n+1} &= \frac{n(n+1)}{2} + (n+1) = \\
 &= \frac{n(n+1)}{2} + \frac{2(n+1)}{2} = \\
 &= \frac{n(n+1) + 2(n+1)}{2} = \\
 &= \frac{(n+2)(n+1)}{2}.
 \end{aligned}$$

□

The important lesson here is that if you're proving something about some inductively defined sequence a_n , induction is the obvious way to go. And even if it isn't (as in the case of the possibilities of dice throws), you can use induction if you can somehow relate the case for $n+1$ to the case for n .

A.2 Strong Induction

In the principle of induction discussed above, we prove $P(0)$ and also if $P(n)$, then $P(n+1)$. In the second part, we assume that $P(n)$ is true and use this assumption to prove $P(n+1)$. Equivalently, of course, we could assume $P(n-1)$ and use it to prove $P(n)$ —the important part is that we be able to carry out the inference from any number to its successor; that we can prove the claim in question for any number under the assumption it holds for its predecessor.

There is a variant of the principle of induction in which we don't just assume that the claim holds for the predecessor $n-1$ of n , but for all numbers smaller than n , and use this assumption to establish the claim for n . This also gives us the claim $P(k)$ for all $k \in \mathbb{N}$. For once we have established $P(0)$, we have thereby established that P holds for all numbers less than 1. And if we know that if $P(l)$ for all $l < n$ then $P(n)$, we know this in particular for $n = 1$. So we can conclude $P(2)$. With this we have proved $P(0)$, $P(1)$, $P(2)$, i.e., $P(l)$ for all $l < 3$, and since we have also the conditional, if $P(l)$ for all $l < 3$, then $P(3)$, we can conclude $P(3)$, and so on.

In fact, if we can establish the general conditional “for all n , if $P(l)$ for all $l < n$, then $P(n)$,” we do not have to establish $P(0)$ anymore, since it follows from it. For remember that a general claim like “for all $l < n$, $P(l)$ ” is true if there are no $l < n$. This is a case of vacuous quantification: “all As are Bs ” is true if there are no As , $\forall x (A(x) \rightarrow B(x))$ is true if no x satisfies $A(x)$. In this case, the formalized version would be “ $\forall l (l < n \rightarrow P(l))$ ”—and that is true if there are no $l < n$. And if $n = 0$ that's exactly the case: no $l < 0$, hence “for all $l < 0$, $P(l)$ ” is true, whatever P is. A proof of “if $P(l)$ for all $l < n$, then $P(n)$ ” thus automatically establishes $P(0)$.

This variant is useful if establishing the claim for n can't be made to just rely on the claim for $n-1$ but may require the assumption that it is true for one or more $l < n$.

A.3 Inductive Definitions

In logic we very often define kinds of objects *inductively*, i.e., by specifying rules for what counts as an object of the kind to be defined which explain how to get new objects of that kind from old objects of that kind. For instance, we often define special kinds of sequences of symbols, such as the terms and formulas of a language, by induction. For a simpler example, consider strings of parentheses, such as “ $()$ ” or “ $()()$ ”. In the second string, the parentheses “balance,” in the first one, they don’t. The shortest such expression is “ $()$ ”. Actually, the very shortest string of parentheses in which every opening parenthesis has a matching closing parenthesis is “”, i.e., the empty sequence \emptyset . If we already have a parenthesis expression p , then putting matching parentheses around it makes another balanced parenthesis expression. And if p and p' are two balanced parentheses expressions, writing one after the other, “ pp' ” is also a balanced parenthesis expression. In fact, any sequence of balanced parentheses can be generated in this way, and we might use these operations to *define* the set of such expressions. This is an *inductive definition*.

Definition A.3. The set of *parexpressions* is inductively defined as follows:

1. \emptyset is a parexpression.
2. If p is a parexpression, then so is (p) .
3. If p and p' are parexpressions $\neq \emptyset$, then so is pp' .
4. Nothing else is a parexpression.

(Note that we have not yet proved that every balanced parenthesis expression is a parexpression, although it is quite clear that every parexpression is a balanced parenthesis expression.)

The key feature of inductive definitions is that if you want to prove something about all parexpressions, the definition tells you which cases you must consider. For instance, if you are told that

q is a parexpression, the inductive definition tells you what q can look like: q can be \emptyset , it can be (p) for some other parexpression p , or it can be pp' for two parexpressions p and $p' \neq \emptyset$. Because of clause (4), those are all the possibilities.

When proving claims about all of an inductively defined set, the strong form of induction becomes particularly important. For instance, suppose we want to prove that for every parexpression of length n , the number of $($ in it is $n/2$. This can be seen as a claim about all n : for every n , the number of $($ in any parexpression of length n is $n/2$.

Proposition A.4. *For any n , the number of $($ in a parexpression of length n is $n/2$.*

Proof. To prove this result by (strong) induction, we have to show that the following conditional claim is true:

If for every $k < n$, any parexpression of length k has $k/2$ $($'s, then any parexpression of length n has $n/2$ $($'s.

To show this conditional, assume that its antecedent is true, i.e., assume that for any $k < n$, parexpressions of length k contain $k/2$ $($'s. We call this assumption the inductive hypothesis. We want to show the same is true for parexpressions of length n .

So suppose q is a parexpression of length n . Because parexpressions are inductively defined, we have three cases: (1) q is \emptyset , (2) q is (p) for some parexpression p , or (3) q is pp' for some parexpressions p and $p' \neq \emptyset$.

1. q is \emptyset . Then $n = 0$, and the number of $($ in q is also 0. Since $0 = 0/2$, the claim holds.
2. q is (p) for some parexpression p . Since q contains two more symbols than p , $\text{len}(p) = n - 2$, in particular, $\text{len}(p) < n$, so the inductive hypothesis applies: the number of $($ in p is $\text{len}(p)/2$. The number of $($ in q is $1 +$ the number of $($

in p , so $= 1 + \text{len}(p)/2$, and since $\text{len}(p) = n - 2$, this gives $1 + (n - 2)/2 = n/2$.

3. q is pp' for some parexpression p and $p' \neq \emptyset$. Since neither p nor $p' = \emptyset$, both $\text{len}(p)$ and $\text{len}(p') < n$. Thus the inductive hypothesis applies in each case: The number of $($ in p is $\text{len}(p)/2$, and the number of $($ in p' is $\text{len}(p')/2$. On the other hand, the number of $($ in q is obviously the sum of the numbers of $($ in p and p' , since $q = pp'$. Hence, the number of $($ in q is $\text{len}(p)/2 + \text{len}(p')/2 = (\text{len}(p) + \text{len}(p'))/2 = \text{len}(pp')/2 = n/2$.

In each case, we've shown that the number of $($ in q is $n/2$ (on the basis of the inductive hypothesis). By strong induction, the proposition follows. \square

A.4 Structural Induction

So far we have used induction to establish results about all natural numbers. But a corresponding principle can be used directly to prove results about all elements of an inductively defined set. This is often called *structural* induction, because it depends on the structure of the inductively defined objects.

Generally, an inductive definition is given by (a) a list of “initial” elements of the set and (b) a list of operations which produce new elements of the set from old ones. In the case of parexpressions, for instance, the initial object is \emptyset and the operations are

$$\begin{aligned} o_1(p) &= (p) \\ o_2(q, q') &= qq' \end{aligned}$$

You can even think of the natural numbers \mathbb{N} themselves as being given by an inductive definition: the initial object is 0, and the operation is the successor function $x + 1$.

In order to prove something about all elements of an inductively defined set, i.e., that every element of the set has a property P , we must:

1. Prove that the initial objects have P
2. Prove that for each operation o , if the arguments have P , so does the result.

For instance, in order to prove something about all parexpressions, we would prove that it is true about \emptyset , that it is true of (p) provided it is true of p , and that it is true about qq' provided it is true of q and q' individually.

Proposition A.5. *The number of (equals the number of) in any parexpression p .*

Proof. We use structural induction. Parexpressions are inductively defined, with initial object \emptyset and the operations o_1 and o_2 .

1. The claim is true for \emptyset , since the number of (in $\emptyset = 0$ and the number of) in \emptyset also $= 0$.
2. Suppose the number of (in p equals the number of) in p . We have to show that this is also true for (p) , i.e., $o_1(p)$. But the number of (in (p) is $1 +$ the number of (in p . And the number of) in (p) is $1 +$ the number of) in p , so the claim also holds for (p) .
3. Suppose the number of (in q equals the number of), and the same is true for q' . The number of (in $o_2(p, p')$, i.e., in pp' , is the sum of the number (in p and p' . The number of) in $o_2(p, p')$, i.e., in pp' , is the sum of the number of) in p and p' . The number of (in $o_2(p, p')$ equals the number of) in $o_2(p, p')$.

The result follows by structural induction. □

APPENDIX B

Biographies

B.1 Georg Cantor

An early biography of Georg Cantor (GAY-org KAHN-tor) claimed that he was born and found on a ship that was sailing for Saint Petersburg, Russia, and that his parents were unknown. This, however, is not true; although he was born in Saint Petersburg in 1845.

Cantor received his doctorate in mathematics at the University of Berlin in 1867. He is known for his work in set theory, and is credited with founding set theory as a distinctive research discipline. He was the first to prove that there are infinite sets of different sizes. His theories, and especially his theory of infinities, caused much debate among mathematicians at the time, and his work was controversial.

Cantor's religious beliefs and his mathematical work were inextricably tied; he even claimed that the theory of transfinite numbers had been communicated to him directly by God. In later



Fig. B.1: Georg Cantor

life, Cantor suffered from mental illness. Beginning in 1984, and more frequently towards his later years, Cantor was hospitalized. The heavy criticism of his work, including a falling out with the mathematician Leopold Kronecker, led to depression and a lack of interest in mathematics. During depressive episodes, Cantor would turn to philosophy and literature, and even published a theory that Francis Bacon was the author of Shakespeare's plays.

Cantor died on January 6, 1918, in a sanitorium in Halle.

Further Reading For full biographies of Cantor, see [Dauben \(1990\)](#) and [Grattan-Guinness \(1971\)](#). Cantor's radical views are also described in the BBC Radio 4 program *A Brief History of Mathematics* ([Sautoy, 2014](#)). If you'd like to hear about Cantor's theories in rap form, see [Rose \(2012\)](#).

B.2 Alonzo Church

Alonzo Church was born in Washington, DC on June 14, 1903. In early childhood, an air gun incident left Church blind in one eye. He finished preparatory school in Connecticut in 1920 and began his university education at Princeton that same year. He completed his doctoral studies in 1927. After a couple years abroad, Church returned to Princeton. Church was known exceedingly polite and careful. His blackboard writing was immaculate, and he would preserve important papers by carefully covering them in Duco cement. Outside of his academic pursuits, he enjoyed reading science fiction magazines and was not afraid to write to the editors if he spotted any inaccuracies in the writing.



Fig. B.2: Alonzo Church

Church's academic achievements were great. Together with his students Stephen Kleene and Barkley Rosser, he developed a theory of effective calculability, the lambda calculus, independently of Alan Turing's development of the Turing machine. The two definitions of computability are equivalent, and give rise to what is now known as the *Church-Turing Thesis*, that a function of the natural numbers is effectively computable if and only if it is computable via Turing machine (or lambda calculus). He also proved what is now known as *Church's Theorem*: The decision problem for the validity of first-order formulas is unsolvable.

Church continued his work into old age. In 1967 he left Princeton for UCLA, where he was professor until his retirement in 1990. Church passed away on August 1, 1995 at the age of 92.

Further Reading For a brief biography of Church, see [Enderton \(N.D.\)](#). Church's original writings on the lambda calculus and the Entscheidungsproblem (Church's Thesis) are [Church \(1936a,b\)](#). [Aspray \(1984\)](#) records an interview with Church about the Princeton mathematics community in the 1930s. Church wrote a series of book reviews of the *Journal of Symbolic Logic* from 1936 until 1979. They are all archived on John MacFarlane's website ([MacFarlane, 2015](#)).

B.3 Gerhard Gentzen

Gerhard Gentzen is known primarily as the creator of structural proof theory, and specifically the creation of the natural deduction and sequent calculus proof systems. He was born on November 24, 1909 in Greifswald, Germany. Gerhard was homeschooled for three years before attending preparatory school, where he was behind most of his classmates in terms of educa-



Fig. B.3: Gerhard Gentzen

where he was behind most of his classmates in terms of educa-

tion. Despite this, he was a brilliant student and showed a strong aptitude for mathematics. His interests were varied, and he, for instance, also wrote poems for his mother and plays for the school theatre.

Gentzen began his university studies at the University of Greifswald, but moved around to Göttingen, Munich, and Berlin. He received his doctorate in 1933 from the University of Göttingen under Hermann Weyl. (Paul Bernays supervised most of his work, but was dismissed from the university by the Nazis.) In 1934, Gentzen began work as an assistant to David Hilbert. That same year he developed the sequent calculus and natural deduction proof systems, in his papers *Untersuchungen über das logische Schließen I–II* [*Investigations Into Logical Deduction I–II*]. He proved the consistency of the Peano axioms in 1936.

Gentzen's relationship with the Nazis is complicated. At the same time his mentor Bernays was forced to leave Germany, Gentzen joined the university branch of the SA, the Nazi paramilitary organization. Like many Germans, he was a member of the Nazi party. During the war, he served as a telecommunications officer for the air intelligence unit. However, in 1942 he was released from duty due to a nervous breakdown. It is unclear whether or not Gentzen's loyalties lay with the Nazi party, or whether he joined the party in order to ensure academic success.

In 1943, Gentzen was offered an academic position at the Mathematical Institute of the German University of Prague, which he accepted. However, in 1945 the citizens of Prague revolted against German occupation. Soviet forces arrived in the city and arrested all the professors at the university. Because of his membership in Nazi organizations, Gentzen was taken to a forced labour camp. He died of malnutrition while in his cell on August 4, 1945 at the age of 35.

Further Reading For a full biography of Gentzen, see [Menzler-Trott \(2007\)](#). An interesting read about mathematicians under Nazi rule, which gives a brief note about Gentzen's life, is given by

Segal (2014). Gentzen's papers on logical deduction are available in the original German (Gentzen, 1935a,b). English translations of Gentzen's papers have been collected in a single volume by Szabo (1969), which also includes a biographical sketch.

B.4 Kurt Gödel

Kurt Gödel (GER-dle) was born on April 28, 1906 in Brünn in the Austro-Hungarian empire (now Brno in the Czech Republic). Due to his inquisitive and bright nature, young Kurt was often called “Der kleine Herr Warum” (Little Mr. Why) by his family. He excelled in academics from primary school onward, where he got less than the highest grade only in mathematics. Gödel was often absent from school due to poor health and was exempt from physical education. Gödel was diagnosed with rheumatic fever during his childhood. Throughout his life, he believed this permanently affected his heart despite medical assessment saying otherwise.



Fig. B.4: Kurt Gödel

Gödel began studying at the University of Vienna in 1920 and completed his doctoral studies in 1929. He first intended to study physics, but his interests soon moved to mathematics and especially logic, in part due to the influence of the philosopher Rudolf Carnap. His dissertation, written under the supervision of Hans Hahn, proved the completeness theorem of first-order predicate logic with identity. Only a couple years later, his most famous results were published—the first and second incompleteness theorems (Gödel, 1931). During his time in Vienna, Gödel was also involved with the Vienna Circle, a group of scientifically-minded philosophers.

In 1938, Gödel married Adele Nimbursky. His parents were not pleased: not only was she six years older than him and already divorced, but she worked as a dancer in a nightclub. Social pressures did not affect Gödel, however, and they remained happily married until his death.

After Nazi Germany annexed Austria in 1938, Gödel and Adele emigrated to the United States, where he took up a position at the Institute for Advanced Study in Princeton, New Jersey. Despite his introversion and eccentric nature, Gödel's time at Princeton was collaborative and fruitful. He published essays in set theory, philosophy and physics. Notably, he struck up a particularly strong friendship with his colleague at the IAS, Albert Einstein.

In his later years, Gödel's mental health deteriorated. His wife's hospitalization in 1977 meant she was no longer able to cook his meals for him. Succumbing to both paranoia and anorexia, and deathly afraid of being poisoned, Gödel refused to eat. He died of starvation on January 14, 1978 in Princeton.

Further Reading For a complete biography of Gödel's life is available, see [Dawson Jr \(1997\)](#). For further biographical pieces, as well as essays about Gödel's contributions to logic and philosophy, see [Wang \(1990\)](#), [Baaz et al. \(2011\)](#), [Takeuti et al. \(2003\)](#), and [Sigmund et al. \(2007\)](#).

Gödel's PhD thesis is available in the original German ([Gödel, 1929](#)). The original text of the incompleteness theorems is ([Gödel, 1931](#)). All of Gödel's published and unpublished writings, as well as a selection of correspondence, are available in English in his *Collected Papers* [Feferman et al. \(1986, 1990\)](#).

For a detailed treatment of Gödel's incompleteness theorems, see [Smith \(2013\)](#). For an informal, philosophical discussion of Gödel's theorems, see Mark Linsenmayer's podcast ([Linsenmayer, 2014](#)).

The Kurt Gödel society keeps Gödel's memory alive by promoting research in logic and other areas influenced by his works

Beckmann and Preining (2004).

B.5 Emmy Noether

Emmy Noether was born in Erlangen, Germany, on March 23, 1882, to an upper-middle class scholarly family. Hailed as the “mother of modern algebra,” Noether made groundbreaking contributions to both mathematics and physics, despite significant barriers to women’s education. In Germany at the time, young girls were meant to be educated in arts and were not allowed to attend college preparatory schools. However, after auditing classes at the Universities of Göttin-



Fig. B.5: Emmy Noether

gen and Erlangen (where her father was professor of mathematics), Noether was eventually able to enrol as a student at Erlangen in 1904, when their policy was updated to allow female students. She received her doctorate in mathematics in 1907.

Despite her qualifications, Noether experienced much resistance during her career. From 1908–1915, she taught at Erlangen without pay. During this time, she caught the attention of David Hilbert, one of the world’s foremost mathematicians of the time, who invited her to Göttingen. However, women were prohibited from obtaining professorships, and she was only able to lecture under Hilbert’s name, again without pay. During this time she proved what is now known as Noether’s theorem, which is still used in theoretical physics today. Noether was finally granted the right to teach in 1919. Hilbert’s response to continued resistance of his university colleagues reportedly was: “Gentlemen, the faculty senate is not a bathhouse.”

Noether was Jewish, and when the Nazis came to power in

1933, she was dismissed from her position. Luckily, Noether was able to emigrate to the United States for a temporary position at Bryn Mawr, Pennsylvania. During her time there she also lectured at Princeton, although she found the university to be unwelcoming to women (Dick, 1981, 81). In 1935, Noether underwent an operation to remove a uterine tumour. She died from an infection as a result of the surgery, and was buried at Bryn Mawr.

Further Reading For a biography of Noether, see Dick (1981). The Perimeter Institute for Theoretical Physics has their lectures on Noether's life and influence available online (Institute, 2015). If you're tired of reading, *Stuff You Missed in History Class* has a podcast on Noether's life and influence (Frey and Wilson, 2015). The collected works of Noether are available in the original German Jacobson (1983).

B.6 Bertrand Russell

Bertrand Russell is hailed as one of the founders of modern analytic philosophy. Born May 18, 1872, Russell was not only known for his work in philosophy and logic, but wrote many popular books in various subject areas. He was also an ardent political activist throughout his life.

Russell was born in Trellech, Monmouthshire, Wales. His parents were members of the British nobility. They were free-thinkers, and even made friends with the radicals in Boston at the time. Unfortunately, Russell's parents died when he was young, and Russell was sent to live with his grandparents. There, he was given a religious upbringing (something

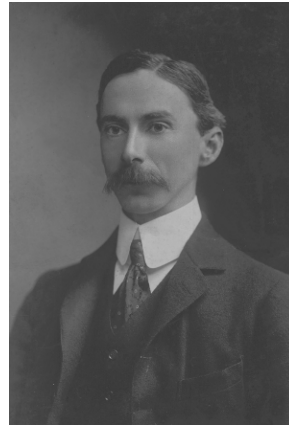


Fig. B.6: Bertrand Russell

his parents had wanted to avoid at all costs). His grandmother was very strict in all matters of morality. During adolescence he was mostly homeschooled by private tutors.

Russell's influence in analytic philosophy, and especially logic, is tremendous. He studied mathematics and philosophy at Trinity College, Cambridge, where he was influenced by the mathematician and philosopher Alfred North Whitehead. In 1910, Russell and Whitehead published the first volume of *Principia Mathematica*, where they championed the view that mathematics is reducible to logic. He went on to publish hundreds of books, essays and political pamphlets. In 1950, he won the Nobel Prize for literature.

Russell's was deeply entrenched in politics and social activism. During World War I he was arrested and sent to prison for six months due to pacifist activities and protest. While in prison, he was able to write and read, and claims to have found the experience "quite agreeable.". He remained a pacifist throughout his life, and was again incarcerated for attending a nuclear disarmament rally in 1961. He also survived a plane crash in 1948, where the only survivors were those sitting in the smoking section. As such, Russell claimed that he owed his life to smoking. Russell was married four times, but had a reputation for carrying on extra-marital affairs. He died on February 2, 1970 at the age of 97 in Penrhyndeudraeth, Wales.

Further Reading Russell wrote an autobiography in three parts, spanning his life from 1872–1967 (Russell, 1967, 1968, 1969). The Bertrand Russell Research Centre at McMaster University is home of the Bertrand Russell archives. See their website at Duncan (2015), for information on the volumes of his collected works (including searchable indexes), and archival projects. Russell's paper *On Denoting* (Russell, 1905) is a classic of 20th century analytic philosophy.

The Stanford Encyclopedia of Philosophy entry on Russell (Irvine, 2015) has sound clips of Russell speaking on Desire and

Political theory. Many video interviews with Russell are available online. To see him talk about smoking and being involved in a plane crash, e.g., see [Russell \(N.D.\)](#). Some of Russell's works, including his *Introduction to Mathematical Philosophy* are available as free audiobooks on [LibriVox \(n.d.\)](#).

B.7 Alfred Tarski

Alfred Tarski was born on January 14, 1901 in Warsaw, Poland (then part of the Russian Empire). Often described as “Napoleonic,” Tarski was boistrous, talkative, and intense. His energy was often reflected in his lectures—he once set fire to a wastebasket while disposing of a cigarette during a lecture, and was forbidden from lecturing in that building again.

Tarski had a thirst for knowledge from a young age. Although later in life he would tell students that he studied logic because it was the only class in which he got a B, his high school records show that he got A's across the board—even in logic. He studied at the University of Warsaw from 1918 to 1924. Although he first intended to study biology, he became interested in mathematics, philosophy, and logic, as the university was the center of the Warsaw School of Logic and Philosophy. Tarski earned his doctorate in 1924 under the supervision of Stanisław Leśniewski.

Before emigrating to the United States in 1939, Tarski completed some of his most important work while working as a secondary school teacher in Warsaw. His work on logical consequence and logical truth were written during this time. In 1939, Tarski was visiting the United States for a lecture tour. During



Fig. B.7: Alfred Tarski

his visit, Germany invaded Poland, and because of his Jewish heritage, Tarski could not return. His wife and children remained in Poland until the end of the war, but were then able to emigrate to the United States as well. Tarski taught at Harvard, the College of the City of New York, and the Institute for Advanced Study at Princeton, and finally the University of California, Berkeley. There he founded the multidisciplinary program in Logic and the Methodology of Science. Tarski died on October 26, 1983 at the age of 82.

Further Reading For more on Tarski's life, see the biography *Alfred Tarski: Life and Logic* (Feferman and Feferman, 2004). Tarski's seminal works on logical consequence and truth are available in English in Corcoran (1983). All of Tarski's original works have been collected into a four volume series, Tarski (1981).

B.8 Alan Turing

Alan Turing was born in Mailda Vale, London, on June 23, 1912. He is considered the father of theoretical computer science. Turing's interest in the physical sciences and mathematics started at a young age. However, as a boy his interests were not represented well in his schools, where emphasis was placed on literature and classics. Consequently, he did poorly in school and was reprimanded by many of his teachers.

Turing attended King's College, Cambridge as an undergraduate, where he studied mathematics. In 1936 Turing developed (what is now called) the Turing machine as an attempt to precisely define the notion of a computable function and to prove the undecidability of the decision problem.



Fig. B.8: Alan Turing

He was beaten to the result by Alonzo Church, who proved the result via his own lambda calculus. Turing's paper was still published with reference to Church's result. Church invited Turing to Princeton, where he spent 1936–1938, and obtained a doctorate under Church.

Despite his interest in logic, Turing's earlier interests in physical sciences remained prevalent. His practical skills were put to work during his service with the British cryptanalytic department at Bletchley Park during World War II. Turing was a central figure in cracking the cypher used by German Naval communications—the Enigma code. Turing's expertise in statistics and cryptography, together with the introduction of electronic machinery, gave the team the ability to crack the code by creating a de-crypting machine called a “bombe.” His ideas also helped in the creation of the world's first programmable electronic computer, the Colossus, also used at Bletchley park to break the German Lorenz cypher.

Turing was gay. Nevertheless, in 1942 he proposed to Joan Clarke, one of his teammates at Bletchley Park, but broke off the engagement and confessed to her that he was homosexual. He had several lovers throughout his lifetime, although homosexual acts were then criminal offences in the UK. In 1952, Turing's house was burgled by a friend of his lover at the time, and when filing a police report, Turing admitted to having a homosexual relationship, under the impression that the government was on their way to legalizing homosexual acts. This was not true, and he was charged with gross indecency. Instead of going to prison, Turing opted for a hormone treatment that reduced libido. Turing was found dead on June 8, 1954, of a cyanide overdose—most likely suicide. He was given a royal pardon by Queen Elizabeth II in 2013.

Further Reading For a comprehensive biography of Alan Turing, see Hodges (2014). Turing's life and work inspired a play, *Breaking the Code*, which was produced in 1996 for TV starring

Derek Jacobi as Turing. *The Imitation Game*, an Academy Award nominated film starring Benedict Cumberbatch and Kiera Knightley, is also loosely based on Alan Turing's life and time at Bletchley Park Tyldum (2014).

Radiolab (2012) has several podcasts on Turing's life and work. BBC Horizon's documentary *The Strange Life and Death of Dr. Turing* is available to watch online (Sykes, 1992). Theelen (2012) is a short video of a working LEGO Turing Machine—made to honour Turing's centenary in 2012.

Turing's original paper on Turing machines and the decision problem is Turing (1937).

B.9 Ernst Zermelo

Ernst Zermelo was born on July 27, 1871 in Berlin, Germany. He had five sisters, though his family suffered from poor health and only three made it to adulthood. His parents also passed away when he was young, leaving him and his siblings orphans when he was seventeen. Zermelo had a deep interest in the arts, and especially in poetry. He was known for being sharp, witty, and critical. His most celebrated mathematical achievements include the introduction of the axiom of choice (in 1904), and his axiomatization of set theory (in 1908).



Fig. B.9: Ernst Zermelo

Zermelo's interests at university were varied. He took courses in physics, mathematics, and philosophy. Under the supervision of Hermann Schwarz, Zermelo completed his dissertation *Investigations in the Calculus of Variations* in 1894 at the University of Berlin. In 1897, he decided to pursue more studies at the Univer-

sity of Göttingen, where he was heavily influenced by the foundational work of David Hilbert. In 1899 he became eligible for professorship, but did not get one until eleven years later—possibly due to his strange demeanour and “nervous haste.”

Zermelo finally received a paid professorship at the University of Zurich in 1910, but was forced to retire in 1916 due to tuberculosis. After his recovery, he was given an honorary professorship at the University of Freiburg in 1921. During this time he worked on foundational mathematics. He became irritated with the works of Thoralf Skolem and Kurt Gödel, and publicly criticized their approaches in his papers. He was dismissed from his position at Freiburg in 1935, due to his unpopularity and his opposition to Hitler’s rise to power in Germany.

The later years of Zermelo’s life were marked by isolation. After his dismissal in 1935, he abandoned mathematics. He moved to the country where he lived modestly. He married in 1944, and became completely dependent on his wife as he was going blind. Zermelo lost his sight completely by 1951. He passed away in Günterstal, Germany, on May 21, 1953.

Further Reading For a full biography of Zermelo, see [Ebbinghaus \(2015\)](#). Zermelo’s seminal 1904 and 1908 papers are available to read in the original German ([Zermelo, 1904, 1908](#)). Zermelo’s collected works, including his writing on physics, are available in English translation in [Ebbinghaus et al. \(2010\)](#); [Ebbinghaus and Kanamori \(2013\)](#).

APPENDIX C

Problems

Problems for Chapter 0

Problems for Chapter 1

Problem 1.1. Show that there is only one empty set, i.e., show that if X and Y are sets without members, then $X = Y$.

Problem 1.2. List all subsets of $\{a, b, c, d\}$.

Problem 1.3. Show that if X has n elements, then $\wp(X)$ has 2^n elements.

Problem 1.4. Prove rigorously that if $X \subseteq Y$, then $X \cup Y = Y$.

Problem 1.5. Prove rigorously that if $X \subseteq Y$, then $X \cap Y = X$.

Problem 1.6. Prove in detail that $X \cup (X \cap Y) = X$. Then compress it into a “textbook proof.” (Hint: for the $X \cup (X \cap Y) \subseteq X$ direction you will need proof by cases, aka \vee Elim.)

Problem 1.7. List all elements of $\{1, 2, 3\}^3$.

Problem 1.8. Show that if X has n elements, then X^k has n^k elements.

Problems for Chapter 2

Problem 2.1. List the elements of the relation \subseteq on the set $\wp(\{a, b, c\})$.

Problem 2.2. Give examples of relations that are (a) reflexive and symmetric but not transitive, (b) reflexive and anti-symmetric, (c) anti-symmetric, transitive, but not reflexive, and (d) reflexive, symmetric, and transitive. Do not use relations on numbers or sets.

Problem 2.3. Show that if R is a weak partial order on X , then $R^- = R \setminus \text{Id}_X$ is a strict partial order and Rxy iff R^-xy for all $x \neq y$.

Problem 2.4. Consider the less-than-or-equal-to relation \leq on the set $\{1, 2, 3, 4\}$ as a graph and draw the corresponding diagram.

Problem 2.5. Show that the transitive closure of R is in fact transitive.

Problems for Chapter 3

Problem 3.1. Show that if f is bijective, an inverse g of f exists, i.e., define such a g , show that it is a function, and show that it is an inverse of f , i.e., $f(g(y)) = y$ and $g(f(x)) = x$ for all $x \in X$ and $y \in Y$.

Problem 3.2. Show that if $f: X \rightarrow Y$ has an inverse g , then f is bijective.

Problem 3.3. Show that if $g: Y \rightarrow X$ and $g': Y \rightarrow X$ are inverses of $f: X \rightarrow Y$, then $g = g'$, i.e., for all $y \in Y$, $g(y) = g'(y)$.

Problem 3.4. Show that if $f: X \rightarrow Y$ and $g: Y \rightarrow Z$ are both injective, then $g \circ f: X \rightarrow Z$ is injective.

Problem 3.5. Show that if $f: X \rightarrow Y$ and $g: Y \rightarrow Z$ are both surjective, then $g \circ f: X \rightarrow Z$ is surjective.

Problem 3.6. Given $f: X \rightarrow Y$, define the partial function $g: Y \rightarrow X$ by: for any $y \in Y$, if there is a unique $x \in X$ such that $f(x) = y$, then $g(y) = x$; otherwise $g(y) \uparrow$. Show that if f is injective, then $g(f(x)) = x$ for all $x \in \text{dom}(f)$, and $f(g(y)) = y$ for all $y \in \text{ran}(f)$.

Problem 3.7. Suppose $f: X \rightarrow Y$ and $g: Y \rightarrow Z$. Show that the graph of $(g \circ f)$ is $R_f \mid R_g$.

Problems for Chapter 4

Problem 4.1. Give an enumeration of the set of all ordered pairs of positive rational numbers.

Problem 4.2. Recall from your introductory logic course that each possible truth table expresses a truth function. In other words, the truth functions are all functions from $\mathbb{B}^k \rightarrow \mathbb{B}$ for some k . Prove that the set of all truth functions is enumerable.

Problem 4.3. Show that the set of all finite subsets of an arbitrary infinite enumerable set is enumerable.

Problem 4.4. Show that if X and Y are countable, so is $X \cup Y$.

Problem 4.5. A set of positive integers is said to be *cofinite* iff it is the complement of a finite set of positive integers. Let I be the set that contains all the finite and cofinite sets of positive integers. Show that I is enumerable.

Problem 4.6. Show that the countable union of countable sets is countable. That is, whenever X_1, X_2, \dots are sets, and each X_i is countable, then the union $\bigcup_{i=1}^{\infty} X_i$ of all of them is also countable.

Problem 4.7. Show that $\wp(\mathbb{N})$ is uncountable.

Problem 4.8. Show that the set of functions $f: \mathbb{Z}^+ \rightarrow \mathbb{Z}^+$ is uncountable by a direct diagonal argument.

Problem 4.9. Show that the set of all sets of pairs of positive integers is uncountable.

Problem 4.10. Show that \mathbb{N}^ω , the set of infinite sequences of natural numbers, is uncountable.

Problem 4.11. Let P be the set of total functions from the set of positive integers to the set $\{0\}$, and let Q be the set of partial functions from the set of positive integers to the set $\{0\}$. Show that P is countable and Q is not.

Problem 4.12. Let S be the set of all total surjective functions from the set of positive integers to the set $\{0,1\}$. Show that S is uncountable.

Problem 4.13. Show that the set \mathbb{R} of all real numbers is uncountable.

Problem 4.14. Show that if X is equinumerous with U and Y is equinumerous with V , and the intersections $X \cap Y$ and $U \cap V$ are empty, then the unions $X \cup Y$ and $U \cup V$ are equinumerous.

Problem 4.15. Given an enumeration of a set X , show that if X is not finite then it is equinumerous with the positive integers \mathbb{Z}^+ .

Problems for Chapter 5

Problem 5.1. Prove [Lemma 5.10](#).

Problem 5.2. Prove [Proposition 5.11](#) (Hint: Formulate and prove a version of [Lemma 5.10](#) for terms.)

Problem 5.3. Give an inductive definition of the bound variable occurrences along the lines of [Definition 5.17](#).

Problem 5.4. Is N , the standard model of arithmetic, covered? Explain.

Problem 5.5. Let $\mathcal{L} = \{c, f, A\}$ with one constant symbol, one one-place function symbol and one two-place predicate symbol, and let the structure M be given by

1. $|M| = \{1, 2, 3\}$
2. $c^M = 3$
3. $f^M(1) = 2, f^M(2) = 3, f^M(3) = 2$
4. $A^M = \{\langle 1, 2 \rangle, \langle 2, 3 \rangle, \langle 3, 3 \rangle\}$

(a) Let $s(v) = 1$ for all variables v . Find out whether

$$M, s \models \exists x (A(f(z), c) \rightarrow \forall y (A(y, x) \vee A(f(y), x)))$$

Explain why or why not.

(b) Give a different structure and variable assignment in which the formula is not satisfied.

Problem 5.6. Complete the proof of [Proposition 5.35](#).

Problem 5.7. Show that if A is a sentence, $M \models A$ iff *there is* a variable assignment s so that $M, s \models A$.

Problem 5.8. Prove [Proposition 5.37](#).

Problem 5.9. Suppose \mathcal{L} is a language without function symbols. Given a structure M and $a \in |M|$, define $M[a/c]$ to be the structure that is just like M , except that $c^{M[a/c]} = a$. Define $M \models A$ for sentences A by:

1. $A \equiv \perp$: $\text{not } M \models A$.
2. $A \equiv R(d_1, \dots, d_n)$: $M \models A$ iff $\langle d_1^M, \dots, d_n^M \rangle \in R^M$.
3. $A \equiv d_1 = d_2$: $M \models A$ iff $d_1^M = d_2^M$.
4. $A \equiv \neg B$: $M \models A$ iff $\text{not } M \models B$.
5. $A \equiv (B \wedge C)$: $M \models A$ iff $M \models B$ and $M \models C$.

6. $A \equiv (B \vee C)$: $M \models A$ iff $M \models A$ or $M \models B$ (or both).
7. $A \equiv (B \rightarrow C)$: $M \models A$ iff not $M \models B$ or $M \models C$ (or both).
8. $A \equiv \forall x B$: $M \models A$ iff for all $a \in |M|$, $M[a/c] \models B[c/x]$, if c does not occur in B .
9. $A \equiv \exists x B$: $M \models A$ iff there is an $a \in |M|$ such that $M[a/c] \models B[c/x]$, if c does not occur in B .

Let x_1, \dots, x_n be all free variables in A , c_1, \dots, c_n constant symbols not in A , $a_1, \dots, a_n \in |M|$, and $s(x_i) = a_i$.

Show that $M, s \models A$ iff $M[a_1/c_1, \dots, a_n/c_n] \models A[c_1/x_1] \dots [c_n/x_n]$.

Problem 5.10. Suppose that f is a function symbol not in $A(x, y)$. Show that there is a M such that $M \models \forall x \exists y A(x, y)$ iff there is a M' such that $M' \models \forall x A(x, f(x))$.

Problem 5.11. Prove [Proposition 5.40](#)

Problem 5.12. 1. Show that $\Gamma \models \perp$ iff Γ is unsatisfiable.

2. Show that $\Gamma, A \models \perp$ iff $\Gamma \models \neg A$.
3. Suppose c does not occur in A or Γ . Show that $\Gamma \models \forall x A$ iff $\Gamma \models A[c/x]$.

Problems for Chapter 6

Problem 6.1. Find formulas in \mathcal{L}_A which define the following relations:

1. n is between i and j ;
2. n evenly divides m (i.e., m is a multiple of n);
3. n is a prime number (i.e., no number other than 1 and n evenly divides n).

Problem 6.2. Suppose the formula $A(x_1, x_2)$ expresses the relation $R \subseteq |M|^2$ in a structure M . Find formulas that express the following relations:

1. the inverse R^{-1} of R ;
2. the relative product $R \mid R$;

Can you find a way to express R^+ , the transitive closure of R ?

Problem 6.3. Let \mathcal{L} be the language containing a 2-place predicate symbol $<$ only (no other constant symbols, function symbols or predicate symbols— except of course $=$). Let N be the structure such that $|N| = \mathbb{N}$, and $<^N = \{\langle n, m \rangle : n < m\}$. Prove the following:

1. $\{0\}$ is definable in N ;
2. $\{1\}$ is definable in N ;
3. $\{2\}$ is definable in N ;
4. for each $n \in \mathbb{N}$, the set $\{n\}$ is definable in N ;
5. every finite subset of $|N|$ is definable in N ;
6. every co-finite subset of $|N|$ is definable in N (where $X \subseteq \mathbb{N}$ is co-finite iff $\mathbb{N} \setminus X$ is finite).

Problems for Chapter 7

Problem 7.1. Give derivations of the following formulas:

1. $\neg(A \rightarrow B) \rightarrow (A \wedge \neg B)$
2. $\forall x (A(x) \rightarrow B) \rightarrow (\exists y A(y) \rightarrow B)$

Problem 7.2. Prove [Proposition 7.11](#)

Problem 7.3. Prove [Proposition 7.12](#)

Problem 7.4. Prove Proposition 7.18

Problem 7.5. Prove Proposition 7.19.

Problem 7.6. Prove Proposition 7.20.

Problem 7.7. Prove Proposition 7.21.

Problem 7.8. Prove Proposition 7.22.

Problem 7.9. Prove Proposition 7.23.

Problem 7.10. Complete the proof of Theorem 7.26.

Problem 7.11. Prove that $=$ is both symmetric and transitive, i.e., give derivations of $\forall x \forall y (x = y \rightarrow y = x)$ and $\forall x \forall y \forall z ((x = y \wedge y = z) \rightarrow x = z)$

Problem 7.12. Give derivations of the following formulas:

1. $\forall x \forall y ((x = y \wedge A(x)) \rightarrow A(y))$
2. $\exists x A(x) \wedge \forall y \forall z ((A(y) \wedge A(z)) \rightarrow y = z) \rightarrow \exists x (A(x) \wedge \forall y (A(y) \rightarrow y = x))$

Problems for Chapter 8

Problem 8.1. Complete the proof of Proposition 8.2.

Problem 8.2. Complete the proof of Lemma 8.9.

Problem 8.3. Complete the proof of Proposition 8.11.

Problem 8.4. Use Corollary 8.17 to prove Theorem 8.16, thus showing that the two formulations of the completeness theorem are equivalent.

Problem 8.5. In order for a derivation system to be complete, its rules must be strong enough to prove every unsatisfiable set inconsistent. Which of the rules of **LK** were necessary to prove completeness? Are any of these rules not used anywhere in the proof? In order to answer these questions, make a list or diagram that shows which of the rules of **LK** were used in which results that lead up to the proof of **Theorem 8.16**. Be sure to note any tacit uses of rules in these proofs.

Problem 8.6. Prove (1) of **Theorem 8.19**.

Problems for Chapter 9

Problems for Chapter 10

Problem 10.1. Choose an arbitrary input and trace through the configurations of the doubler machine in **Example 10.4**.

Problem 10.2. The double machine in **Example 10.4** writes its output to the right of the input. Come up with a new method for solving the doubler problem which generates its output immediately to the right of the end-of-tape marker. Build a machine that executes your method. Check that your machine works by tracing through the configurations.

Problem 10.3. Design a Turing-machine with alphabet $\{\sqcup, A, B\}$ that accepts any string of A s and B s where the number of A s is the same as the number of B s *and* all the A s precede all the B s, and rejects any string where the number of A s is not equal to the number of B s or the A s do not precede all the B s. (E.g., the machine should accept $AABB$, and $AAABBB$, but reject both AAB and $AABBAABB$.)

Problem 10.4. Design a Turing-machine with alphabet $\{\sqcup, A, B\}$ that takes as input any string α of A s and B s and duplicates them to produce an output of the form $\alpha\alpha$. (E.g. input $ABBA$ should result in output $ABBAABBA$).

Problem 10.5. *Alphabetical?*: Design a Turing-machine with alphabet $\{\sqcup, A, B\}$ that when given as input a finite sequence of As and Bs checks to see if all the As appear left of all the Bs or not. The machine should leave the input string on the tape, and output either halt if the string is “alphabetical”, or loop forever if the string is not.

Problem 10.6. *Alphabetizer*: Design a Turing-machine with alphabet $\{\sqcup, A, B\}$ that takes as input a finite sequence of As and Bs rearranges them so that all the As are to the left of all the Bs. (e.g., the sequence *BABAA* should become the sequence *AAABB*, and the sequence *ABBABB* should become the sequence *ABBBBB*).

Problem 10.7. Trace through the configurations of the machine for input $\langle 3, 5 \rangle$.

Problem 10.8. *Subtraction*: Design a Turing machine that when given an input of two non-empty strings of strokes of length n and m , where $n > m$, computes the function $f(n, m) = n - m$.

Problem 10.9. *Equality*: Design a Turing machine to compute the following function:

$$\text{equality}(x, y) = \begin{cases} 1 & \text{if } x = y \\ 0 & \text{if } x \neq y \end{cases}$$

where x and y are integers greater than 0.

Problem 10.10. Design a Turing machine to compute the function $\min(x, y)$ where x and y are positive integers represented on the tape by strings of I 's separated by a \sqcup . You may use additional symbols in the alphabet of the machine.

The function \min selects the smallest value from its arguments, so $\min(3, 5) = 3$, $\min(20, 16) = 16$, and $\min(4, 4) = 4$, and so on.

Problems for Chapter 11

Problem 11.1. The Three Halting (3-Halt) problem is the problem of giving a decision procedure to determine whether or not an arbitrarily chosen Turing Machine halts for an input of three strokes on an otherwise blank tape. Prove that the 3-Halt problem is unsolvable.

Problem 11.2. Show that if the halting problem is solvable for Turing machine and input pairs M_e and n where $e \neq n$, then it is also solvable for the cases where $e = n$.

Problem 11.3. We proved that the halting problem is unsolvable if the input is a number e , which identifies a Turing machine M_e via an enumeration of all Turing machines. What if we allow the description of Turing machines from [section 11.2](#) directly as input? (This would require a larger alphabet of course.) Can there be a Turing machine which decides the halting problem but takes as input descriptions of Turing machines rather than indices? Explain why or why not.

Problem 11.4. Complete case (3) of the proof of [Lemma 11.9](#).

Photo Credits

Georg Cantor, p. 180: Portrait of Georg Cantor by Otto Zeth courtesy of the Universitätsarchiv, Martin-Luther Universität Halle–Wittenberg. UAHW Rep. 40-VI, Nr. 3 Bild 102.

Alonzo Church, p. 181: Portrait of Alonzo Church courtesy of the Department of Rare Books and Special Collections, Princeton University Library. Alonzo Church Papers (Cog48), Box 60, Folder 3. © Princeton University.

Gerhard Gentzen, p. 182: Portrait of Gerhard Gentzen playing ping-pong courtesy of Ekhart Mentzler-Trott.

Kurt Gödel, p. 184: Portrait of Kurt Gödel from the Shelby White and Leon Levy Archives Center, Institute for Advanced Study, Princeton, NJ, USA, on deposit at Princeton University Library, Department of Rare Books and Special Collections, Kurt Gödel Papers (Co282), Box 14b, #110,000. The Open Logic Project has obtained permission from the Institute's Archives Center to use this image. Permission from the Archives Center is required for any other use.

Emmy Noether, p. 186: Portrait of Emmy Noether, ca. 1922, courtesy of the Abteilung für Handschriften und Seltene Drucke, Niedersächsische Staats- und Universitätsbibliothek Göttingen, Cod. Ms. D. Hilbert 754, Bl. 14 Nr. 73.

Bertrand Russell, p. 187: Portrait of Bertrand Russell, ca. 1907, courtesy of the William Ready Division of Archives and Research Collections, McMaster University Library. Bertrand Russell Archives,

Box 2, f. 4.

Alfred Tarski, p. 189: Passport photo of Alfred Tarski, 1939, courtesy of the Bancroft Library, University of California, Berkeley. Alfred Tarski Papers, Banc MSS 84/49.

Alan Turing, p. 190: Alan Mathison Turing by Elliott & Fry, 29 March 1951, NPG x82217, © National Portrait Gallery, London. Used under a Creative Commons BY-NC-ND 3.0 license.

Ernst Zermelo, p. 192: Portrait of Ernst Zermelo, ca. 1922, courtesy of the Abteilung für Handschriften und Seltene Drucke, Niedersächsische Staats- und Universitätsbibliothek Göttingen, Cod. Ms. D. Hilbert 754, Bl. 6 Nr. 25.

Bibliography

- Aspray, William. 1984. The Princeton mathematics community in the 1930s: Alonzo Church. URL http://www.princeton.edu/mudd/finding_aids/mathoral/pmc05.htm. Interview.
- Baaz, Matthias, Christos H. Papadimitriou, Hilary W. Putnam, Dana S. Scott, and Charles L. Harper Jr. 2011. *Kurt Gödel and the Foundations of Mathematics: Horizons of Truth*. Cambridge: Cambridge University Press.
- Beckmann, Arnold and Norbert Preining. 2004. Kurt Gödel Society. URL <http://kgs.logic.at/index.php?id=2>.
- Church, Alonzo. 1936a. A note on the Entscheidungsproblem. *Journal of Symbolic Logic* 1: 40–41.
- Church, Alonzo. 1936b. An unsolvable problem of elementary number theory. *American Journal of Mathematics* 58: 345–363.
- Corcoran, John. 1983. *Logic, Semantics, Metamathematics*. Indianapolis: Hackett, 2nd ed.
- Dauben, Joseph. 1990. *Georg Cantor: His Mathematics and Philosophy of the Infinite*. Princeton: Princeton University Press.
- Dawson Jr, John. 1997. *Logical Dilemmas: The Life and Work of Kurt Gödel*. Boca Raton: CRC Press.


- Dick, Auguste. 1981. *Emmy Noether 1882–1935*. Boston: Birkhäuser.
- Duncan, Arlene. 2015. The Bertrand Russell Research Centre. URL <http://russell.mcmaster.ca/>.
- Ebbinghaus, Heinz-Dieter. 2015. *Ernst Zermelo: An Approach to his Life and Work*. Berlin: Springer-Verlag.
- Ebbinghaus, Heinz-Dieter, Craig G. Fraser, and Akihiro Kanamori. 2010. *Ernst Zermelo. Collected Works*, vol. 1. Berlin: Springer-Verlag.
- Ebbinghaus, Heinz-Dieter and Akihiro Kanamori. 2013. *Ernst Zermelo: Collected Works*, vol. 2. Berlin: Springer-Verlag.
- Enderton, Herbert B. N.D. *Alonzo Church: Life and Work*. Cambridge: MIT Press.
- Feferman, Anita and Solomon Feferman. 2004. *Alfred Tarski: Life and Logic*. Cambridge: Cambridge University Press.
- Feferman, Solomon, John W. Dawson Jr., Stephen C. Kleene, Gregory H. Moore, Robert M. Solovay, and Jean van Heijenoort. 1986. *Kurt Gödel: Collected Works. Vol. 1: Publications 1929–1936*. Oxford: Oxford University Press.
- Feferman, Solomon, John W. Dawson Jr., Stephen C. Kleene, Gregory H. Moore, Robert M. Solovay, and Jean van Heijenoort. 1990. *Kurt Gödel: Collected Works. Vol. 2: Publications 1938–1974*. Oxford: Oxford University Press.
- Frey, Holly and Tracy V. Wilson. 2015. Stuff you Missed in History Class: Emmy Noether, Mathematics Trailblazer. URL <http://www.missedinhistory.com/podcasts/emmy-noether-mathematics-trailblazer/>. Podcast audio.
- Gentzen, Gerhard. 1935a. Untersuchungen über das logische Schließen I. *Mathematische Zeitschrift* 39: 176–210. English translation in Szabo (1969), pp. 68–131.

- Gentzen, Gerhard. 1935b. Untersuchungen über das logische Schließen II. *Mathematische Zeitschrift* 39: 176–210, 405–431. English translation in Szabo (1969), pp. 68–131.
- Gödel, Kurt. 1929. Über die Vollständigkeit des Logikkalküls [on the completeness of the calculus of logic]. Dissertation, Universität Wien. Reprinted and translated in Feferman et al. (1986), pp. 60–101.
- Gödel, Kurt. 1931. Über formal unentscheidbare Sätze der *Principia Mathematica* und verwandter Systeme I [On Formally Undecidable Propositions of *Principia Mathematica* and Related Systems I]. *Monatshefte für Mathematik und Physik* 38: 173–198. Reprinted and translated in Feferman et al. (1986), pp. 144–195.
- Grattan-Guinness, Ivor. 1971. Towards a biography of Georg Cantor. *Annals of Science* 27(4): 345–391.
- Hodges, Andrew. 2014. *Alan Turing: The Enigma*. London: Vintage.
- Institute, Perimeter. 2015. Emmy Noether: Her Life, Work, and Influence. URL <https://www.youtube.com/watch?v=tNNyAyMRsgE>. Video Lecture.
- Irvine, Andrew David. 2015. Stanford Encyclopedia of Philosophy: Sound Clips of Bertrand Russell Speaking. URL <http://plato.stanford.edu/entries/russell/russell-soundclips.html>.
- Jacobson, Nathan. 1983. *Emmy Noether: Gesammelte Abhandlungen—Collected Papers*. Berlin: Springer-Verlag.
- LibriVox. n.d. LibriVox - Bertrand Russell. URL https://librivox.org/author/1508?primary_key=1508&search_category=author&search_page=1&search_form=get_results. Collection of public domain audiobooks.

- Linsenmayer, Mark. 2014. The Partially Examined Life: Gödel on Math. URL <http://www.partiallyexaminedlife.com/2014/06/16/ep95-godel/>. Podcast audio.
- MacFarlane, John. 2015. Alonzo Church's JSL Reviews. URL <http://johnmacfarlane.net/church.html>.
- Menzler-Trott, Eckart. 2007. *Logic's Lost Genius: The Life of Gerhard Gentzen*. Providence: American Mathematical Society.
- Radiolab. 2012. The Turing Problem. URL <http://www.radiolab.org/story/193037-turing-problem/>. Podcast audio.
- Rose, Daniel. 2012. A Song About Georg Cantor. URL <https://www.youtube.com/watch?v=QUP5Z4Fb5k4>. Audio Recording.
- Russell, Bertrand. 1905. On denoting. *Mind* 14: 479–493.
- Russell, Bertrand. 1967. *The Autobiography of Bertrand Russell*, vol. 1. London: Allen and Unwin.
- Russell, Bertrand. 1968. *The Autobiography of Bertrand Russell*, vol. 2. London: Allen and Unwin.
- Russell, Bertrand. 1969. *The Autobiography of Bertrand Russell*, vol. 3. London: Allen and Unwin.
- Russell, Bertrand. N.D. Bertrand Russell on Smoking. URL https://www.youtube.com/watch?v=80oLTiVW_lc. Video Interview.
- Sautoy, du Marcus. 2014. A Brief History of Mathematics: Georg Cantor. URL <http://www.bbc.co.uk/programmes/b00ss1j0>. Audio Recording.
- Segal, Sanford L. 2014. *Mathematicians under the Nazis*. Princeton: Princeton University Press.

- Sigmund, Karl, John Dawson, Kurt Mühlberger, Hans Magnus Enzensberger, and Juliette Kennedy. 2007. Kurt Gödel: Das Album—The Album. *The Mathematical Intelligencer* 29(3): 73–76.
- Smith, Peter. 2013. *An Introduction to Gödel's Theorems*. Cambridge: Cambridge University Press.
- Sykes, Christopher. 1992. BBC Horizon: The strange life and death of Dr. Turing. URL <https://www.youtube.com/watch?v=gyusnGbBSHE>.
- Szabo, M. E. 1969. *The Collected Papers of Gerhard Gentzen*. Amsterdam: North-Holland.
- Takeuti, Gaisi, Nicholas Passell, and Mariko Yasugi. 2003. *Memoirs of a Proof Theorist: Gödel and Other Logicians*. Singapore: World Scientific.
- Tarski, Alfred. 1981. *The Collected Works of Alfred Tarski*, vol. I–IV. Basel: Birkhäuser.
- Theelen, Andre. 2012. Lego turing machine. URL <https://www.youtube.com/watch?v=FTSAiF9AHN4>.
- Turing, Alan M. 1937. On computable numbers, with an application to the “Entscheidungsproblem”. *Proceedings of the London Mathematical Society, 2nd Series* 42: 230–265.
- Tyldum, Morten. 2014. The Imitation Game. Motion picture.
- Wang, Hao. 1990. *Reflections on Kurt Gödel*. Cambridge: MIT Press.
- Zermelo, Ernst. 1904. Beweis, daß jede Menge wohlgeordnet werden kann. *Mathematische Annalen* 59: 514–516. English translation in (Ebbinghaus et al., 2010, pp. 115–119).

Zermelo, Ernst. 1908. Untersuchungen über die Grundlagen der Mengenlehre I. *Mathematische Annalen* 65(2): 261–281. English translation in (Ebbinghaus et al., 2010, pp. 189–229).



About the Open Logic Project

The *Open Logic Text* is an open-source, collaborative textbook of formal meta-logic and formal methods, starting at an intermediate level (i.e., after an introductory formal logic course). Though aimed at a non-mathematical audience (in particular, students of philosophy and computer science), it is rigorous.

The *Open Logic Text* is a collaborative project and is under active development. Coverage of some topics currently included may not yet be complete, and many sections still require substantial revision. We plan to expand the text to cover more topics in the future. We also plan to add features to the text, such as a glossary, a list of further reading, historical notes, pictures, better explanations, sections explaining the relevance of results to philosophy, computer science, and mathematics, and more problems and examples. If you find an error, or have a suggestion, please let the project team know.

The project operates in the spirit of open source. Not only is the text freely available, we provide the LaTeX source under

the Creative Commons Attribution license, which gives anyone the right to download, use, modify, re-arrange, convert, and redistribute our work, as long as they give appropriate credit.

Please see the Open Logic Project website at openlogicproject.org for additional information.