Logic and Proof

Release 0.1

Jeremy Avigad, Robert Y. Lewis, and Floris van Doorn

Oct 15, 2017

CONTENTS

CHAPTER

ONE

INTRODUCTION

1.1 Mathematical Proof

Although there is written evidence of mathematical activity in Egypt as early as 3000 BC, many scholars locate the birth of mathematics proper in ancient Greece around the sixth century BC, when deductive proof was first introduced. Aristotle credited Thales of Miletus with recognizing the importance of not just what we know but how we know it, and finding grounds for knowledge in the deductive method. Around 300 BC, Euclid codified a deductive approach to geometry in his treatise, the *Elements*. Through the centuries, Euclid's axiomatic style was held as a paradigm of rigorous argumentation, not just in mathematics, but in philosophy and the sciences as well.

Here is an example of an ordinary proof, in contemporary mathematical language. It establishes a fact that was known to the Pythagoreans.

Theorem. $\sqrt{2}$ is irrational, which is to say, it cannot be expressed as a fraction a/b, where a and b are integers.

Proof. Suppose $\sqrt{2} = a/b$ for some pair of integers a and b. By removing any common factors, we can assume a/b is in lowest terms, so that a and b have no factor in common. Then we have $a = \sqrt{2}b$, and squaring both sides, we have $a^2 = 2b^2$.

The last equation implies that a^2 is even, and since the square of an odd number is odd, a itself must be even as well. We therefore have $a = 2c$ for some integer c. Substituting this into the equation $a^2 = 2b^2$, we have $4c^2 = 2b^2$, and hence $2c^2 = b^2$. This means that b^2 is even, and so b is even as well.

The fact that a and b are both even contradicts the fact that a and b have no common factor. So the original assumption that $\sqrt{2} = a/b$ is false.

In the next example, we focus on the natural numbers,

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

A natural number n greater than or equal to 2 is said to be *composite* if it can be written as a product $n = m \cdot k$ where neither m nor k is equal to 1, and *prime* otherwise. Notice that if $n = m \cdot k$ witnesses the fact that n is composite, then m and k are both smaller than n. Notice also that, by convention, 0 and 1 are considered neither prime nor composite.

Theorem. Every natural number greater than equal to 2 can be written as a product of primes.

Proof. We proceed by induction on n. Let n be any natural number greater than 2. If n is prime, we are done; we can consider n itself as a product with one term. Otherwise, n is composite, and we can write $n = m \cdot k$ where m and k are smaller than n and greater than 1. By the inductive hypothesis, each of m and k can be written as a product of primes, say $m = p_1 \cdot p_2 \cdot \ldots \cdot p_u$ and $k = q_1 \cdot q_2 \cdot \ldots \cdot q_v$. But then we have

$$n = m \cdot k = p_1 \cdot p_2 \cdot \ldots \cdot p_u \cdot q_1 \cdot q_2 \cdot \ldots \cdot q_v,$$

a product of primes, as required.

Later, we will see that more is true: every natural number greater than 2 can be written as a product of primes in a unique way, a fact known as the *fundamental theorem of arithmetic*.

The first goal of this course is to teach you to write clear, readable mathematical proofs. We will do this by considering a number of examples, but also by taking a reflective point of view: we will carefully study the components of mathematical language and the structure of mathematical proofs, in order to gain a better understanding of how they work.

1.2 Symbolic Logic

Towards understanding how proofs work, it will be helpful to study a subject known as "symbolic logic," which provides an idealized model of mathematical language and proof. In the *Prior Analytics*, the ancient Greek philosopher set out to analyze patterns of reasoning, and developed the theory of the *syllogism*. Here is one instance of a syllogism:

Every man is an animal.

Every animal is mortal.

Therefore every man is mortal.

Aristotle observed that the correctness of this inference has nothing to do with the truth or falsity of the individual statements, but, rather, the general pattern:

Every A is B.

Every B is C.

Therefore every A is C.

We can substitute various properties for A, B, and C; try substituting the properties of being a fish, being a unicorn, being a swimming creature, being a mythical creature, etc. The various statements that result may come out true or false, but all the instantiations will have the following crucial feature: if the two hypotheses come out true, then the conclusion comes out true as well. We express this by saying that the inference is *valid*.

Although the patterns of language addressed by Aristotle's theory of reasoning are limited, we have him to thank for a crucial insight: we can classify valid patterns of inference by their logical form, while abstracting away specific content. It is this fundamental observation that underlies the entire field of symbolic logic.

In the seventeenth century, Leibniz proposed the design of a *characteristica universalis*, a universal symbolic language in which one would express any assertion in a precise way, and a *calculus ratiocinatur*, a “calculus of thought” which would express the precise rules of reasoning. Leibniz himself took some steps to develop such a language and calculus, but much greater strides were made in the nineteenth century, through the work of Boole, Frege, Peirce, Schroeder, and others. Early in the twentieth century, these efforts blossomed into the field of mathematical logic.

If you consider the examples of proofs in the last section, you will notice that some terms and rules of inference are specific to the subject matter at hand, having to do with numbers, and the properties of being prime, composite, even, odd, and so on. But there are other terms and rules of inference that are not domain specific, such as those related to the words “every,” “some,” “and,” and “if … then.” The goal of symbolic logic is to identify these core elements of reasoning and argumentation and explain how they work, as well as to explain how more domain-specific notions are introduced and used.

To that end, we will introduce symbols for key logical notions, including the following:

- $A \to B$, “if A then B“
- $A \wedge B$, “A and B“
- $A \vee B$, “A or B“
- $\neg A$, “not A“
- $\forall x\ A$, “for every x, A“
- $\exists x\ A$, “for some x, A“

We will then provide a formal proof system that will let us establish, deductively, that certain entailments between such statements are valid.

The proof system we will use is a version of *natural deduction*, a type of proof system introduced by Gerhard Gentzen in the 1930’s to model informal styles of argument. In this system, the fundamental unit of judgment is the assertion that an assertion, A, follows from a finite set of hypotheses, Γ. This is written as $\Gamma \vdash A$. If Γ and Δ are two finite sets of hypotheses, we will write Γ, Δ for the *union* of these two sets, that is, the set consisting of all the hypotheses in each. With these conventions, the rule for the conjunction symbol can be expressed as follows:

$$\frac{\Gamma \vdash A \qquad \Delta \vdash B}{\Gamma, \Delta \vdash A \wedge B}$$

This should be interpreted as saying: assuming A follows from the hypotheses Γ, and B follows from the hypotheses Δ, $A \wedge B$ follows from the hypotheses in both Γ and Δ.

We will see that one can write such proofs more compactly leaving the hypotheses implicit, so that the rule above is expressed as follows:

$$\frac{A \qquad B}{A \wedge B}$$

In this format, a snippet of the first proof in the previous section might be rendered as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\neg even(b)} \qquad \dfrac{\forall x\ (\neg even(x) \to \neg even(x^2))}{\neg even(b) \to \neg even(b^2))}}{\neg even(b^2)} \qquad even(b^2)}{\bot}}{even(b)}$$

The complexity of such proofs can quickly grow out of hand, and complete proofs of even elementary mathematical facts can become quite long. Such systems are not designed for writing serious mathematics.

Rather, they provide idealized models of mathematical inference, and insofar as they capture something of the structure of an informal proof, they enable us to study the properties of mathematical reasoning.

The second goal of this course is to help you understand natural deduction, as an example of a formal deductive system.

1.3 Interactive Theorem Proving

Early work in mathematical logic aimed to show that ordinary mathematical arguments could be modeled in symbolic calculi, at least in principle. As noted above, complexity issues limit the range of what can be accomplished in practice; even elementary mathematical arguments require long derivations that are hard to write and hard to read, and do little to promote understanding of the underlying mathematics.

Since the end of the twentieth century, however, the advent of computational proof assistants has begun to make complete formalization feasible. Working interactively with theorem proving software, users can construct formal derivations of complex theorems that can be stored and checked by computer. Automated methods can be used to fill in small gaps by hand, verify long calculations axiomatically, or fill in long chains of inferences deterministically. The reach of automation is currently fairly limited, however. The strategy used in interactive theorem proving is to ask users to provide just enough information for the system to be able to construct and check a formal derivation. This typically involves writing proofs in a sort of "programming language" that is designed with that purpose in mind. For example, here is a short proof in the *Lean* theorem prover:

```
section
variables (P Q : Prop)

theorem my_theorem : P ∧ Q → Q ∧ P :=
assume h : P ∧ Q,
have P, from and.left h,
have Q, from and.right h,
show Q ∧ P, from and.intro ‹Q› ‹P›

end
```

If you are reading the present text in online form, you will find a button underneath the formal "proof script" that says "try it!" Pressing the button copies the proof to an editor window at right, and runs a version of Lean inside your browser to process the proof, turn it into an axiomatic derivation, and verify its correctness. You can experiment by varying the text in the editor and pressing the "play" button to see the result.

Proofs in Lean can access a library of prior mathematical results, all verified down to axiomatic foundations. A goal of the field of interactive theorem proving is to reach the point where any contemporary theorem can be verified in this way. For example, here is a formal proof that the square root of two is irrational, following the model of the informal proof presented above:

```
import data.nat.prime
open nat

theorem sqrt_two_irrational {a b : ℕ} (co : gcd a b = 1) :
  a^2 ≠ 2 * b^2 :=
assume h : a^2 = 2 * b^2,
have 2 ∣ a^2,
  by simp [h],
have 2 ∣ a,
  from dvd_of_prime_of_dvd_pow prime_two this,
exists.elim this $
assume (c : nat) (aeq : a = 2 * c),
have 2 * (2 * c^2) = 2 * b^2,
  by simp [eq.symm h, aeq]; simp [pow_succ],
have 2 * c^2 = b^2,
  from eq_of_mul_eq_mul_left dec_trivial this,
have 2 ∣ b^2,
  by simp [eq.symm this],
have 2 ∣ b,
  from dvd_of_prime_of_dvd_pow prime_two this,
have 2 ∣ gcd a b,
  from dvd_gcd ‹2 ∣ a› ‹2 ∣ b›,
have 2 ∣ (1 : ℕ),
  by simp * at *,
show false, from absurd ‹2 ∣ 1› dec_trivial
```

The third goal of this course is to teach you to write elementary proofs in Lean. The facts that we will ask you to prove in Lean will be more elementary than the informal proofs we will ask you to write, but our intent is that formal proofs will model and clarify the informal proof strategies we will teach you.

1.4 The Semantic Point of View

As we have presented the subject here, the goal of symbolic logic is to specify a language and rules of inference that enable us to get at the truth in a reliable way. The idea is that the symbols we choose denote objects and concepts that have a fixed meaning, and the rules of inference we adopt enable us to draw true conclusions from true hypotheses.

One can adopt another view of logic, however, as a system where some symbols have a fixed meaning, such as the symbols for "and," "or," and "not," and others have a meaning that is taken to vary. For example, the expression $P \wedge (Q \vee R)$, read "P and either Q or R," may be true or false *depending on the basic assertions that P, Q, and R stand for.* More precisely, the truth of the compound expression depends only on whether the component symbols denote expressions that are true or false. For example, if P, Q, and R stand for "seven is prime," "seven is even," and "seven is odd," respectively, then the expression is true. If we replace "seven" by "six," the statement is false. More generally, the expression comes out true whenever P is true and at least one of Q and R is true, and false otherwise.

From this perspective, logic is not so much a language for asserting truth, but a language for describing possible states of affairs. In other words, logic provides a specification language, with expressions that can be true or false depending on how we interpret the symbols that are allowed to vary. For example, if we fix the meaning of the basic predicates, the statement "there is a red block between two blue blocks" may be true or false of a given "world" of blocks, and we can take the expression to describe the set of worlds in which it is true. Such a view of logic is important in computer science, where we use logical expressions to select entries from a database matching certain criteria, to specify properties of hardware and software systems, or to specify constraints that we would like a constraint solver to satisfy.

There are important connections between the syntactic / deductive point of view on the one hand, and the semantic / model-theoretic point of view on the other. We will explore some of these along the way. For example, we will see that it is possible to view the "valid" assertions as those that are true under all possible interpretations of the non-fixed symbols, and the "valid" inferences as those that maintain truth in all possible states and affairs. From this point of view, a deductive system should only allow us to derive valid assertions and entailments, a property known as *soundness*. If a deductive system is strong enough to allow us to verify *all* valid assertions and entailments, it is said to be *complete*.

The fourth goal of course is to convey the semantic view of logic, and understand how logical expressions can be used to specify states of affairs.

1.5 Goals Summarized

To summarize, these are the goals of this course:

- to teach you to write clear, “literate,” mathematical proofs
- to introduce you to symbolic logic and the formal modeling of deductive proof
- to introduce you to interactive theorem proving
- to teach you to understand how to use logic as a precise language for making claims about systems of objects and the relationships between them, and specifying certain states of affairs.

Let us take a moment to consider the relationship between some of these goals. It is important not to confuse the first three. We are dealing with three kinds of mathematical language: ordinary mathematical language, the symbolic representations of mathematical logic, and computational implementations in interactive proof assistants. These are very different things!

Symbolic logic is not meant to replace ordinary mathematical language, and you should not use symbols like $\wedge$ and $\vee$ in ordinary mathematical proofs any more than you would use them in place of the words “and” and “or” in letters home to your parents. Natural languages provide nuances of expression that can convey levels of meaning and understanding that go beyond pattern matching to verify correctness. At the same time, modeling mathematical language with symbolic expressions provides a level of precision that makes it possible to turn mathematical language itself into an object of study. Each has its place, and we hope to get you to appreciate the value of each without confusing the two.

The proof languages used by interactive theorem provers lie somewhere between the two extremes. On the one hand, they have to be specified with enough precision for a computer to process them and act appropriately; on the other hand, they aim to capture some of the higher-level nuances and features of informal language in a way that enables us to write more complex arguments and proofs. Rooted in symbolic logic and designed with ordinary mathematical language in mind, they aim to bridge the gap between the two.

1.6 About this Textbook

Both this online textbook and the *Lean* theorem prover it invokes are new and ongoing projects, and in places they are still rough. Please bear with us! Your feedback will be quite helpful.

CHAPTER

TWO

PROPOSITIONAL LOGIC

2.1 A Puzzle

The following puzzle, titled “Malice and Alice,” is from George J. Summers’ *Logical Deduction Puzzles*.

Alice, Alice’s husband, their son, their daughter, and Alice’s brother were involved in a murder. One of the five killed one of the other four. The following facts refer to the five people mentioned:

1. A man and a woman were together in a bar at the time of the murder.
2. The victim and the killer were together on a beach at the time of the murder.
3. One of Alice’s two children was alone at the time of the murder.
4. Alice and her husband were not together at the time of the murder.
5. The victim’s twin was not the killer.
6. The killer was younger than the victim.

Which one of the five was the victim?

Take some time to try to work out a solution. (You should assume that the victim’s twin is one of the five people mentioned.) Summers’ book offers the following hint: “First find the locations of two pairs of people at the time of the murder, and then determine who the killer and the victim were so that no condition is contradicted.”

2.2 A Solution

If you have worked on the puzzle, you may have noticed a few things. First, it is helpful to draw a diagram, and to be systematic about searching for an answer. The number of characters, locations, and attributes is finite, so that there are only finitely many possible “states of affairs” that need to be considered. The numbers are also small enough so that systematic search through all the possibilities, though tedious, will eventually get you to the right answer. This is a special feature of logic puzzles like this; you would not expect to show, for example, that every even number greater than two can be written as a sum of primes by running through all the possibilities.

Another thing that you may have noticed is that the question seems to presuppose that there is a unique answer to the question, which is to say, of all the states of affairs that meet the list of conditions, there is only one person who can possibly be the killer. *A priori*, without that assumption, there is a difference between finding *some* person who could have been the victim, and show that that person *had* to be the victim. In other words, there is a difference between exhibiting some state of affairs that meets the criteria, and demonstrating conclusively that no other solution is possible.

The published solution in the book not only produces a state of affairs that meets the criterion, but at the same time proves that this is the only one that does so. It is quoted below, in full.

From (1), (2), and (3), the roles of the five people were as follows: Man and Woman in the bar, Killer and Victim on the beach, and Child alone.

Then, from (4), either Alice's husband was in the bar and Alice was on the beach, or Alice was in the bar and Alice's husband was on the beach.

If Alice's husband was in the bar, the woman he was with was his daughter, the child who was alone was his son, and Alice and her brother were on the beach. Then either Alice or her brother was the victim; so the other was the killer. But, from (5), the victim had a twin, and this twin was innocent. Since by Alice and her brother could only be twins to each other, this situation is impossible. Therefore Alice's husband was not in the bar.

So Alice was in the bar. If Alice was in the bar, she was with her brother or her son.

If Alice was with her brother, her husband was on the beach with one of the two children. From (5), the victim could not be her husband, because none of the others could be his twin; so the killer was her husband and the victim was the child he was with. But this situation is impossible, because it contradicts (6). Therefore, Alice was not with her brother in the bar.

So Alice was with her son in the bar. Then the child who was alone was her daughter. Therefore, Alice's husband was with Alice's brother on the beach. From previous reasoning, the victim could not be Alice's husband. But the victim could be Alice's brother because Alice could be his twin.

So *Alice's brother was the victim* and Alice's husband was the killer.

This argument relies on some "extralogical" elements, for example, that a father cannot be younger than his child, and that a parent and his or her child cannot be twins. But the argument also involves a number of common logical terms and associated patterns of inference. In the next section, we will focus on some of the key logical terms occurring in the argument above, words like "and," "or," "not," and "if … then."

Our goal is to give an account of the patterns of inference that govern the use of those terms. To that end, using the methods of symbolic logic, we will introduce variables A, B, C, … to stand for fundamental statements, or *propositions*, and symbols $\wedge$, $\vee$, $\neg$, and $\to$ to stand for "and," "or," "not," and "if … then … ," respectively. Doing so will let us focus on the way that compound statements are built up from basic ones using the logical terms, while abstracting away from the specific content. We will also adopt a stylized notation for representing inferences as *rules*: the inscription

$$\frac{A \qquad B}{C}$$

indicates that statement C is a *logical consequence* of A and B.

2.3 Rules of Inference

2.3.1 Implication

The first pattern of inference we will discuss, involving the "if … then …" construct, can be hard to discern. Its use is largely implicit in the solution above. The inference in the fourth paragraph, spelled out in greater detail, runs as follows:

If Alice was in the bar, Alice was with her brother or her son.

Alice was in the bar.

Alice was with her brother or son.

This rule is sometimes known as *modus ponens*, or "implication elimination," since it tells us how to use an implication in an argument. As a rule, it is expressed as follows:

$$\frac{A \to B \qquad A}{B} \to\mathrm{E}$$

Read this as saying that if you have a proof of $A \to B$, possibly from some hypotheses, and a proof of A, possibly from hypotheses, then combining these yields a proof of B, from the hypotheses in both subproofs.

The rule for deriving an "if … then" statement is more subtle. Consider the beginning of the third paragraph, which argues that if Alice's husband was in the bar, then Alice or her brother was the victim. Abstracting away some of the details, the argument has the following form:

Suppose Alice's husband was in the bar.

Then …

Then …

Then Alice or her brother was the victim.

Thus, if Alice's husband was in the bar, then Alice or her brother was the victim.

This is a form of *hypothetical reasoning*. On the supposition that A holds, we argue that B holds as well. If we are successful, we have shown that A implies B, without supposing A. In other words, the temporary assumption that A holds is "canceled" by making it explicit in the conclusion.

$$\begin{array}{c} \overline{A}^{\,1} \\ \vdots \\ \dfrac{B}{A \to B}\,1 \to\mathrm{I} \end{array}$$

The hypothesis is given the label 1; when the introduction rule is applied, the label 1 indicates the relevant hypothesis. The line over the hypothesis indicates that the assumption has been "canceled" by the introduction rule.

2.3.2 Conjunction

As was the case for implication, other logical connectives are generally characterized by their *introduction* and *elimination* rules. An introduction rule shows how to establish a claim involving the connective, while an elimination rule shows how to use such a statement that contains the connective to derive others.

Let us consider, for example, the case of conjunction, that is, the word "and." Informally, we establish a conjunction by establishing each conjunct. For example, informally we might argue:

Alice's brother was the victim.

Alice's husband was the killer.

Therefore Alice's brother was the victim and Alice's husband was the killer.

The inference seems almost too obvious to state explicitly, since the word "and" simply combines the two assertions into one. Informal proofs often downplay the distinction. In symbolic logic, the rule reads as follows:

$$\frac{A \qquad B}{A \wedge B}\,\wedge\mathrm{I}$$

The two elimination rules allow us to extract the two components:

Alice's husband was in the bar and Alice was on the beach.

So Alice's husband was in the bar.

Or:

Alice's husband was in the bar and Alice was on the beach.

So Alice's was on the beach.

In symbols, these patterns are rendered as follows:

$$\frac{A \wedge B}{A}\,\wedge\mathrm{E_l} \qquad \frac{A \wedge B}{B}\,\wedge\mathrm{E_r}$$

Here the l and r stand for "left" and "right".

2.3.3 Negation and Falsity

In logical terms, showing "not A" amounts to showing that A leads to a contradiction. For example:

Suppose Alice's husband was in the bar.

...

This situation is impossible.

Therefore Alice's husband was not in the bar.

This is another form of hypothetical reasoning, similar to that used in establishing an "if … then" statement: we temporarily assume A, show that leads to a contradiction, and conclude that "not A" holds. In symbols, the rule reads as follows:

$$\cfrac{\cfrac{}{A}\,1 \\ \vdots \\ \bot}{\neg A}\,1\ \neg\mathrm{I}$$

The elimination rule is dual to these. It expresses that if we have both "A" and "not A," then we have a contradiction. This pattern is illustrated in the informal argument below, which is implicit in the fourth paragraph of the solution to "Malice and Alice."

The killer was Alice's husband and the victim was the child he was with.

So the killer was not younger than his victim.

But according to (6), the killer was younger than his victim.

This situation is impossible.

In symbolic logic, the rule of inference is expressed as follows:

$$\frac{\neg A \qquad A}{\bot}\ \neg\mathrm{E}$$

Notice also that in the symbolic framework, we have introduced a new symbol, $\bot$. It corresponds to natural language phrases like "this is a contradiction" or "this is impossible."

What are the rules governing $\bot$? In the proof system we will introduce in the next chapter, there is no introduction rule; "false" is false, and there should be no way to prove it, other than extract it from contradictory hypotheses. On the other hand, the system provides a rule that allows us to conclude anything from a contradiction:

$$\frac{\bot}{A}\ \bot\mathrm{E}$$

The elimination rule also has the fancy Latin name, *ex falso sequitur quodlibet*, which means "anything you want follows from falsity."

This elimination rule is harder to motivate from a natural language perspective, but, nonetheless, it is needed to capture common patterns of inference. One way to understand it is this. Consider the following statement:

For every natural number n, if n is prime and greater than 2, then n is odd.

We would like to say that this is a true statement. But if it is true, then it is true of any particular number n. Taking $n = 2$, we have the statement:

If 2 is prime and greater than 2, then 2 is odd.

In this conditional statement, both the antecedent and succedent are false. The fact that we are committed to saying that this statement is true shows that we should be able to prove, one way or another, that the statement 2 is odd follows from the false statement that 2 is prime and greater than 2. The *ex falso* neatly encapsulates this sort of inference.

Notice that if we define $\neg A$ to be $A \to \bot$, then the rules for negation introduction and elimination are nothing more than implication introduction and elimination, respectively. We can think of $\neg A$ expressed colorfully by saying "if A is true, then pigs have wings," where "pigs have wings" is stands for $\bot$.

Having introduced a symbol for "false," it is only fair to introduce a symbol for "true." In contrast to "false," "true" has no elimination rule, only an introduction rule:

$$\frac{}{\top}$$

Put simply, "true" is true.

2.3.4 Disjunction

The introduction rules for disjunction, otherwise known as "or," are straightforward. For example, the claim that condition (3) is met in the proposed solution can be justified as follows:

Alice's daughter was alone at the time of the murder.

Therefore, either Alice's daughter was alone at the time of the murder, or Alice's son was alone at the time of the murder.

In symbolic terms, the two introduction rules are as follows:

$$\dfrac{A}{A \vee B}\,{\vee}\mathrm{I_l} \qquad \dfrac{B}{A \vee B}\,{\vee}\mathrm{I_r}$$

Here, again, the l and r stand for "left" and "right".

The disjunction elimination rule is trickier, but it represents a natural form of case-based hypothetical reasoning. The instances that occur in the solution to "Malice and Alice" are all special cases of this rule, so it will be helpful to make up a new example to illustrate the general phenomenon. Suppose, in the argument above, we had established that either Alice's brother or her son was in the bar, and we wanted to argue for the conclusion that her husband was on the beach. One option is to argue by cases: first, consider the case that her brother was in the bar, and argue for the conclusion on the basis of that assumption; then consider the case that her son was in the bar, and argue for the same conclusion, this time on the basis of the second assumption. Since the two cases are exhaustive, if we know that the conclusion holds in each case, we know that it holds outright. The pattern looks something like this:

Either Alice's brother was in the bar, or Alice's son was in the bar.

Suppose, in the first case, that her brother was in the bar. Then … Therefore, her husband was on the beach.

On the other hand, suppose her son was in the bar. In that case, … Therefore, in this case also, her husband was on the beach.

Either way, we have established that her husband was on the beach.

In symbols, this pattern is expressed as follows:

$$\dfrac{A \vee B \qquad \begin{matrix}\overline{A}^{\,1} \\ \vdots \\ C\end{matrix} \qquad \begin{matrix}\overline{B}^{\,1} \\ \vdots \\ C\end{matrix}}{C}\;1\ {\vee}\mathrm{E}$$

What makes this pattern confusing is that it requires two instances of nested hypothetical reasoning: in the first block of parentheses, we temporarily assume A, and in the second block, we temporarily assume B. When the dust settles, we have established C outright.

There is another pattern of reasoning that is commonly used with "or," as in the following example:

Either Alice's husband was in the bar, or Alice was in the bar.

Alice's husband was not in the bar.

So Alice was in the bar.

In symbols, we would render this rule as follows:

$$\frac{A \vee B \qquad \neg A}{B}$$

We will see in the next chapter that it is possible to *derive* this rule from the others. As a result, we will *not* take this to be a fundamental rule of inference in our system.

2.3.5 If and only if

In mathematical arguments, it is common to say of two statements, A and B, that "A holds if and only if B holds." This assertion is sometimes abbreviated "A iff B," and means simply that A implies B and B implies A. It is not essential that we introduce a new symbol into our logical language to model this connective, since the statement can be expressed, as we just did, in terms of "implies" and "and." But notice that the length of the expression doubles because A and B are each repeated. The logical abbreviation is therefore convenient, as well as natural.

The conditions of "Malice and Alice" imply that Alice is in the bar if and only if Alice's husband is on the beach. Such a statement is established by arguing for each implication in turn:

I claim that Alice is in the bar if and only if Alice's husband is on the beach.

To see this, first suppose that Alice is in the bar.

Then …

Hence Alice's husband is on the beach.

Conversely, suppose Alice's husband is on the beach.

Then …

Hence Alice is in the bar.

Notice that with this example, we have varied the form of presentation, stating the conclusion first, rather than at the end of the argument. This kind of "signposting" is common in informal arguments, in that is helps guide the reader's expectations and foreshadow where the argument is going. The fact that formal systems of deduction do not generally model such nuances marks a difference between formal and informal arguments, a topic we will return to below.

The introduction is modeled in natural deduction as follows:

$$\frac{\begin{matrix}\overline{A}^{\,1} & \overline{B}^{\,1}\\ \vdots & \vdots \\ B & A\end{matrix}}{A \leftrightarrow B}\,{\scriptstyle 1}\ \leftrightarrow\mathrm{I}$$

The elimination rules for iff are unexciting. In informal language, here is the "left" rule:

Alice is in the bar if and only if Alice's husband is on the beach.

Alice is in the bar.

Hence, Alice's husband is on the beach.

The "right" rule simply runs in the opposite direction.

Alice is in the bar if and only if Alice's husband is on the beach.

Alice's husband is on the beach.

Hence, Alice is in the bar.

Rendered in natural deduction, the rules are as follows:

$$\frac{A \leftrightarrow B \qquad A}{B} \leftrightarrow\mathrm{E}_l \qquad \frac{A \leftrightarrow B \qquad B}{A} \leftrightarrow\mathrm{E}_r$$

2.3.6 Proof by Contradiction

We saw an example of an informal argument that implicitly uses the introduction rule for negation:

Suppose Alice's husband was in the bar.

...

This situation is impossible.

Therefore Alice's husband was not in the bar.

Consider the following argument:

Suppose Alice's husband was not on the beach.

...

This situation is impossible.

Therefore Alice's husband was on the beach.

At first glance, you might think this argument follows the same pattern as the one before. But a closer look should reveal a difference: in the first argument, a negation is *introduced* into the conclusion, whereas in the second, it is *eliminated* from the hypothesis. Using negation introduction to close the second argument would yield the conclusion "It is not the case that Alice's husband was not on the beach." The rule of inference that replaces the conclusion with the positive statement that Alice's husband *was* on the beach is called a *proof by contradiction*. (It also has a fancy name, *reductio ad absurdum*, "reduction to an absurdity.")

It may be hard to see the difference between the two rules, because we commonly take the statement "Alice's husband was not not on the beach" to be a roundabout and borderline ungrammatical way of saying that Alice's husband was on the beach. Indeed, the rule is equivalent to adding an axiom that says that for every statement A, "not not A" is equivalent to A.

There is a style of doing mathematics known as "constructive mathematics" that denies the equivalence of "not not A" and A. Constructive arguments tend to have much better computational interpretations; a proof that something is true should provide explicit evidence that the statement is true, rather than evidence that it can't possibly be false. We will discuss constructive reasoning in a later chapter. Nonetheless, proof by contradiction is used extensively in contemporary mathematics, and so, in the meanwhile, we will use proof by contradiction freely as one of our basic rules.

In natural deduction, proof by contradiction is expressed by the following pattern:

$$\dfrac{\begin{array}{c}\overline{\neg A}^{\,1}\\ \vdots \\ \bot\end{array}}{A}\ \text{RAA,1}$$

The assumption $\neg A$ is canceled at the final inference.

2.4 The Language of Propositional Logic

The language of propositional logic starts with symbols A, B, C, … which are intended to range over basic assertions, or propositions, which can be true or false. Compound expressions are built up using parentheses and the logical symbols introduced in the last section. For example,

$$((A \wedge \neg B) \to \neg(C \vee D))$$

is an example of a propositional formula.

When writing expressions in symbolic logic, we will adopt the an order of operations which allow us to drop superfluous parentheses. When parsing an expression:

- negation binds most tightly
- then conjunctions and disjunctions, from right to left
- and finally implications and bi-implications.

So, for example, the expression $\neg A \vee B \to C \wedge D$ is understood as $((\neg A) \vee B) \to (C \wedge D)$

For example, suppose we assign the following variables:

- A: Alice's husband was in the bar
- B: Alice was on the beach
- C: Alice was in the bar
- D: Alice's husband was on the beach

Then the statement "either Alice's husband was in the bar and Alice was on the beach, or Alice was in the bar and Alice's husband was on the beach would be rendered as

$$(A \wedge B) \vee (C \wedge D)$$

Sometimes the appropriate translation is not so straightforward, however. Because natural language is more flexible and nuanced, a degree of abstraction and regimentation is needed to carry out the translation. Sometimes different translations are arguably reasonable. In happy situations, alternative translations will be logically equivalent, in the sense that one can derive each from the other using purely logical rules. In less happy situations, the translations will not be equivalent, in which case the original statement is simply ambiguous, from a logical point of view. In cases like that, choosing a symbolic representation helps clarify the intended meaning.

Consider, for example, a statement like "Alice was with her son on the beach, but her husband was alone." We might choose variables as follows:

- A: Alice was on the beach
- B: Alice's son was on the beach
- C: Alice's husband was alone

In that case, we might represent the statement in symbols as $A \wedge B \wedge C$. Using the word "with" may seem to connote more that the fact that Alice and her son were both on the beach; for example, it seems to connote that they aware of each others' presence, interacting, etc. Similarly, although we have translated the word "but" and "and," the word "but" also convey information; in this case, it seems to emphasize a contrast, while in other situations, it can be used to assert a fact that is contrary to expectations. In both cases, then, the logical rendering models certain features of the original sentence while abstracting others.

2.5 Exercises

1. Here is another (gruesome) logic puzzle by George J. Summers, called "Murder in the Family."

   > Murder occurred one evening in the home of a father and mother and their son and daughter. One member of the family murdered another member, the third member witnessed the crime, and the fourth member was an accessory after the fact.
   >
   > (a) The accessory and the witness were of opposite sex.
   >
   > (b) The oldest member and the witness were of opposite sex.
   >
   > (c) The youngest member and the victim were of opposite sex.
   >
   > (d) The accessory was older than the victim.
   >
   > (e) The father was the oldest member.
   >
   > (f) The murderer was not the youngest member.
   >
   > Which of the four—father, mother, son, or daughter—was the murderer?

   Solve this puzzle, and *write a clear argument* to establish that your answer is correct.

2. Using the mnemonic F (Father), M (Mother), D (Daughter), S (Son), Mu (Murderer), V (Victim), W (Witness), A (Accessory), O (Oldest), Y (Youngest), we can define propositional variables like FM (Father is the Murderer), DV (Daughter is the Victim), FO (Father is Oldest), VY (Victim is Youngest), etc. Notice that only the son or daughter can be the youngest, and only the mother or father can be the oldest.

   With these conventions, the first clue can be represented

   $$((FA \vee SA) \to (MW \vee DW)) \wedge ((MA \vee DA) \to (FW \vee SW)),$$

   in other words, if the father or son was the accessory, then the mother or daughter was the witness, and vice-versa. Represent the other five clues in a similar manner.

   Representing the fourth clue is tricky. Try to write down a formula that describes all the possibilities that are not ruled out by the information.

3. Consider the following three hypotheses:

   - Alan likes kangaroos, and either Betty likes frogs or Carl likes hamsters.
   - If Betty likes frogs, then Alan doesn't like kangaroos.
   - If Carl likes hamsters, then Betty likes frogs.

   Write a clear argument to show that these three hypotheses are contradictory.

CHAPTER

THREE

NATURAL DEDUCTION FOR PROPOSITIONAL LOGIC

Reflecting on the arguments in the previous chapter, we see that, intuitively speaking, some inferences are *valid* and some or not. For example, if, in a chain of reasoning, we had established "A and B," it would seem perfectly reasonable to conclude B. If we had established A, B, and "If A and B then C," it would be reasonable to conclude C. On the other hand, if we had established "A or B," we would not be justified concluding B without further information.

The task of symbolic logic is to develop a precise mathematical theory that explains which inferences are valid and why. There are two general approaches to spelling out the notion of validity. In this chapter, we will consider the *deductive* approach: an inference is valid if it can be justified by fundamental rules of reasoning that reflect the meaning of the logical terms involved. In Chapter 6 we will consider the "semantic" approach: an inference is valid if it is an instance of a pattern that always yields a true conclusion from true hypotheses.

3.1 Derivations in Natural Deduction

We have seen that the language of propositional logic allows us to build up expressions from propositional variables $A, B, C, \ldots$ using propositional connectives like $\to$, $\wedge$, $\vee$, and $\neg$. We will now consider a formal deductive system that we can use to *prove* propositional formulas. There are a number of such systems on offer; the one will use is called *natural deduction*, designed by Gerhard Gentzen in the 1930's.

In natural deduction, every proof is a proof from *hypotheses*. In other words, in any proof, there is a finite set of hypotheses $\{B, C, \ldots\}$ and a conclusion A, and what the proof shows is that A follows from $B, C, \ldots$.

Like formulas, proofs are built by putting together smaller proofs, according to the rules. For instance, the way to read the and-introduction rule,

$$\frac{A \qquad B}{A \wedge B}$$

is as follows: if you have a proof P_1 of A from some hypotheses, and you have a proof P_2 of B from some hypotheses, then you can put them together using this rule to obtain a proof of $A \wedge B$, which uses all the hypotheses in P_1 together with all the hypotheses in P_2. For example, this is a proof of $(A \wedge B) \wedge (A \wedge C)$ from three hypotheses, A, B, and C:

$$\frac{\dfrac{A \qquad B}{A \wedge B} \qquad \dfrac{A \qquad C}{A \wedge C}}{(A \wedge B) \wedge (A \wedge C)}$$

One thing that makes natural deduction confusing is that when you put together proofs in this way, hypotheses can be eliminated, or, as we will say, *canceled*. For example, we can apply the implies-introduction rule to the last proof, and obtain the following proof of $B \to (A \wedge B) \wedge (A \wedge C)$ from only *two* hypotheses, A and C:

$$\dfrac{\dfrac{\dfrac{A \qquad \overline{B}^{\,1}}{A \wedge B} \qquad \dfrac{A \qquad C}{A \wedge C}}{(A \wedge B) \wedge (A \wedge C)}}{B \to (A \wedge B) \wedge (A \wedge C)}\;1$$

Here, we have used the label 1 to indicate the place where the hypothesis B was canceled. Any label will do, though we will tend to use numbers for that purpose.

We can continue to cancel the hypothesis A:

$$\dfrac{\dfrac{\dfrac{\dfrac{\overline{A}^{\,2} \qquad \overline{B}^{\,1}}{A \wedge B} \qquad \dfrac{\overline{A}^{\,2} \qquad C}{A \wedge C}}{(A \wedge B) \wedge (A \wedge C)}}{B \to (A \wedge B) \wedge (A \wedge C)}\;1}{A \to (B \to (A \wedge B) \wedge (A \wedge C))}\;2$$

The result is a proof using only the hypothesis C. We can continue to cancel that hypothesis as well:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\overline{A}^{\,2} \qquad \overline{B}^{\,1}}{A \wedge B} \qquad \dfrac{\overline{A}^{\,2} \qquad \overline{C}^{\,3}}{A \wedge C}}{(A \wedge B) \wedge (A \wedge C)}}{B \to (A \wedge B) \wedge (A \wedge C)}\;1}{A \to (B \to (A \wedge B) \wedge (A \wedge C))}\;2}{C \to (A \to (B \to (A \wedge B) \wedge (A \wedge C)))}\;3$$

The resulting proof uses no hypothesis at all. In other words, it establishes the conclusion outright.

Notice that in the second step, we canceled two "copies" of the hypothesis A. In natural deduction, we can choose which hypotheses to cancel; we could have canceled either one, and left the other hypothesis *open*. In fact, we can also carry out the implication-introduction rule and cancel *zero* hypotheses. For example, the following is a short proof of $A \to B$ from the hypothesis B:

$$\dfrac{B}{A \to B}$$

In this proof, "zero" copies of A have are canceled.

Also notice that although we are using letters like A, B, and C as propositional variables, in the proofs above we can replace them by any propositional formula. For example, we can replace A by the formula $(D \vee E)$ everywhere, and still have correct proofs. In some presentations of logic, different letters are used for to stand for propositional variables and arbitrary propositional formulas, but we will continue to blur the distinction. You can think of A, B, and C as standing for propositional variables or formulas, as you prefer. If you think of them as propositional variables, just keep in mind that in any rule or proof, you can replace every variable by a different formula, and still have a valid rule or proof.

Finally, notice also that in these examples, we have assumed a special rule as the starting point for building proofs. It is called the assumption rule, and it looks like this:

$$A$$

What it means is that at any point we are free to simply assume a formula, A. The single formula A constitutes a one-line proof, and the way to read this proof is as follows: assuming A, we have proved A.

The remaining rules of inference were given in the last chapter, and we summarize them here.

Implication:

$$\dfrac{\begin{matrix}\overline{A}^{\,1} \\ \vdots \\ B\end{matrix}}{A \to B}\;1\ \to\text{I} \qquad\qquad \dfrac{A \to B \qquad A}{B}\;\to\text{E}$$

Conjunction:

$$\frac{A \qquad B}{A \wedge B}\wedge\mathrm{I} \qquad\qquad \frac{A \wedge B}{A}\wedge\mathrm{E_l} \qquad\qquad \frac{A \wedge B}{B}\wedge\mathrm{E_r}$$

Negation:

$$\frac{\begin{array}{c}\overline{A}^{\,1}\\ \vdots \\ \bot\end{array}}{\neg A}\,1\ \neg\mathrm{I} \qquad\qquad \frac{\neg A \qquad A}{\bot}\neg\mathrm{E}$$

Disjunction:

$$\frac{A}{A \vee B}\vee\mathrm{I_l} \qquad\qquad \frac{B}{A \vee B}\vee\mathrm{I_r} \qquad\qquad \frac{A \vee B \qquad \begin{array}{c}\overline{A}^{\,1}\\ \vdots \\ C\end{array} \qquad \begin{array}{c}\overline{B}^{\,1}\\ \vdots \\ C\end{array}}{C}\,1\ \vee\mathrm{E}$$

Truth and falsity:

$$\frac{\bot}{A}\bot\mathrm{E} \qquad\qquad \frac{}{\top}\top\mathrm{I}$$

Bi-implication:

$$\frac{\begin{array}{c}\overline{A}^{\,1}\\ \vdots \\ B\end{array} \qquad \begin{array}{c}\overline{B}^{\,1}\\ \vdots \\ A\end{array}}{A \leftrightarrow B}\,1\ \leftrightarrow\mathrm{I} \qquad\qquad \frac{A \leftrightarrow B \qquad A}{B}\leftrightarrow\mathrm{E}_l \qquad\qquad \frac{A \leftrightarrow B \qquad B}{A}\leftrightarrow\mathrm{E}_r$$

Reductio ad absurdum (proof by contradiction):

$$\frac{\begin{array}{c}\overline{\neg A}^{\,1}\\ \vdots \\ \bot\end{array}}{A}\,1\ \mathrm{RAA}$$

3.2 Examples

Let us consider some more examples of natural deduction proofs. In each case, you should think about what the formulas say and which rule of inference is invoked at each step. Also pay close attention to which hypotheses are canceled at each stage. If you look at any node of the tree, what has been established at that point is that the claim follows from all the hypotheses above it that haven't been canceled yet.

The following is a proof of $A \to C$ from $A \to B$ and $B \to C$:

$$\frac{\dfrac{\dfrac{\overline{A}^{\,1} \qquad A \to B}{B} \qquad B \to C}{C}}{A \to C}\,1$$

Intuitively, the formula

$$(A \to B) \wedge (B \to C) \to (A \to C)$$

"internalizes" the conclusion of the previous proof. The $\wedge$ symbol is used to combine hypotheses, and the $\to$ symbol is used to express that the right-hand side is a consequence of the left. Here is a proof of that formula:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{1}{A} \quad \dfrac{\dfrac{}{(A\to B)\wedge(B\to C)}{}^{2}}{A\to B}}{B} \quad \dfrac{\dfrac{}{(A\to B)\wedge(B\to C)}{}^{2}}{B\to C}}{C}}{A\to C}{}^{1}}{(A\to B)\wedge(B\to C)\to(A\to C)}{}^{2}$$

The next proof shows that if a conclusion, C, follows from A and B, then it follows from their conjunction.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{A\to(B\to C)}{}^{2} \quad \dfrac{\dfrac{}{A\wedge B}{}^{1}}{A}}{B\to C} \quad \dfrac{\dfrac{}{A\wedge B}{}^{1}}{B}}{C}}{A\wedge B\to C}{}^{1}}{(A\to(B\to C))\to(A\wedge B\to C)}{}^{2}$$

The conclusion of the next proof can be interpreted as saying that if it is not the case that one of A or B is true, then they are both false. It illustrates the use of the rules for negation.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\neg(A\vee B)}{}^{3} \quad \dfrac{\dfrac{}{A}{}^{1}}{A\vee B}}{\bot}}{\neg A}{}^{1} \quad \dfrac{\dfrac{\dfrac{}{\neg(A\vee B)}{}^{3} \quad \dfrac{\dfrac{}{B}{}^{2}}{A\vee B}}{\bot}}{\neg B}{}^{2}}{\neg A\wedge\neg B}}{\neg(A\vee B)\to\neg A\wedge\neg B}{}^{3}$$

Finally, the next two examples illustrate the use of the *ex falso* rule. The first is a derivation of an arbitrary formula B from $\neg A$ and A:

$$\dfrac{\dfrac{\neg A \qquad A}{\bot}}{B}$$

The second shows that B follows from A and $\neg A \vee B$:

$$\dfrac{\neg A\vee B \qquad \dfrac{\dfrac{\dfrac{}{\neg A}{}^{1} \qquad A}{\bot}}{B} \qquad \dfrac{}{B}{}^{1}}{B}{}^{1}$$

In some proof systems, these rules are taken to be part of the system. But we do not need to that with our system: these two examples show that the rules can be *derived* from our other rules.

3.3 Forward and Backward Reasoning

Natural deduction is supposed to represent an idealized model of the patterns of reasoning and argumentation we use, for example, when working with logic puzzles as in the last chapter. There are obvious differences: we describe natural deduction proofs with symbols and two-dimensional diagrams, whereas our informal arguments are written with words and paragraphs. It is worthwhile to reflect on what *is* captured by the model. Natural deduction is supposed to clarify the *form* and *structure* of our logical arguments, describe the appropriate means of justifying a conclusion, and explain the sense in which the rules we use are valid.

Constructing natural deduction proofs can be confusing, but it is helpful to think about *why* it is confusing. We could, for example, decide that natural deduction is not a good model for logical reasoning. Or we might come to the conclusion that the features of natural deduction that make it confusing tell us something interesting about ordinary arguments.

In the "official" description, natural deduction proofs are constructed by putting smaller proofs together to obtain bigger ones. To prove $A \wedge B \to B \wedge A$, we start with the hypothesis $A \wedge B$. Then we construct, separately, the following two proofs:

$$\frac{A \wedge B}{B} \qquad \frac{A \wedge B}{A}$$

Then we use these two proofs to construct the following one:

$$\dfrac{\dfrac{A \wedge B}{B} \qquad \dfrac{A \wedge B}{A}}{B \wedge A}$$

Finally, we apply the implies-introduction rule to this proof to cancel the hypothesis and obtain the desired conclusion:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{A \wedge B}{}^{1}}{B} \qquad \dfrac{\dfrac{}{A \wedge B}{}^{1}}{A}}{B \wedge A}}{A \wedge B \to B \wedge A}{}^{1}$$

The process is similar to what happens in an informal argument, where we start with some hypotheses, and work forward towards a conclusion.

Suppose Susan is tall and John is happy.

Then, in particular, John is happy.

Also, Susan is tall.

So John is happy and Susan is tall.

Therefore we have shown that if Susan is tall and John is happy, then John is happy and Susan is tall.

However, when we *read* natural deduction proofs, we often read them backwards. First, we look at the bottom to see what is being proved. Then we consider the rule that is used to prove it, and see what premises the rule demands. Then we look to see how those claims are proved, and so on. Similarly, when we *construct* a natural deduction proof, we typically work backwards as well: we start with the claim we are trying to prove, put that at the bottom, and look for rules to apply.

At times that process breaks down. Suppose we are left with a goal that is a single propositional variable, A. There are no introduction rules that can be applied, so, unless A is a hypothesis, it has to come from an elimination rule. But that underspecifies the problem: perhaps the A comes from applying the and elimination rule to $A \wedge B$, or from applying the or elimination rule to C and $C \to A$. At that point, we look to the hypotheses, and start working forwards. If, for example, our hypotheses are C and $C \to A \wedge B$, we would then work forward to obtain $A \wedge B$ and A.

There is thus a general heuristic for proving theorems in natural deduction:

1. Start by working backwards from the conclusion, using the introduction rules. For example, if you are trying to prove a statement of the form $A \to B$, add A to your list of hypotheses and try to derive B. If you are trying to prove a statement of the form $A \wedge B$, use the and-introduction rule to reduce your task to proving A, and then proving B.
2. When you have run out things to do in the first step, use elimination rules to work forwards. If you have hypotheses $A \to B$ and A, apply modus ponens to derive B. If you have a hypothesis $A \vee B$, use-or elimination to split on cases, considering A in one case and B in the other.

In Chapter 5 we will add one more element to this list: if all else fails, try a proof by contradiction.

The tension between forward and backward reasoning is found in informal arguments as well, in mathematics and elsewhere. When we prove a theorem, we typically reason forward, using assumptions, hypotheses, definitions, and background knowledge. But we also keep the goal in mind, and that helps us make sense of the forward steps.

When we turn to interactive theorem proving, we will see that *Lean* has mechanisms to support both forward and backward reasoning. These form a bridge between informal styles of argumentation and the natural deduction model, and thereby provide a clearer picture of what is going on.

Another confusing feature of natural deduction proofs is that every hypothesis has a *scope*, which is to say, there are only certain points in the proof where an assumption is available for use. Of course, this is also a feature of informal mathematical arguments. Suppose a paragraph begins "Let x be any number less than 100," argues that x has at most five prime factors, and concludes "thus we have shown that every number less than 100 has at most five factors." The reference "x", and the assumption that it is less than 100, is only active within the scope of the paragraph. If the next paragraph begins with the phrase "Now suppose x is any number greater than 100," then, of course, the assumption that x is less than 100 no longer applies.

In natural deduction, a hypothesis is available from the point where it is assumed until the point where it is canceled. We will see that interactive theorem proving languages also have mechanisms to determine the scope of references and hypotheses, and that these, too, shed light on scoping issues in informal mathematics.

3.4 Reasoning by Cases

The rule for eliminating a disjunction is confusing, but we can make sense of it with an example. Consider the following informal argument:

George is either at home or on campus.

If he is at home, he is studying.

If he is on campus, he is with his friends.

Therefore, George is either studying or with his friends.

Let A be the statement that George is at home, let B be the statement that George is on campus, let C be the statement that George is studying, and let D be the statement the George is with his friends. Then the argument above has the following pattern: from $A \vee B$, $A \to C$, and $B \to D$, conclude $C \vee D$. In natural deduction, we can not get away with drawing this conclusion in a single step, but it does not take too much work to flesh it out into a proper proof. Informally, we have to argue as follows.

Georges is either at home or on campus.

> Case 1: Suppose he is at home. We know that if he is at home, then he is studying. So, in this case, he is studying. Therefore, in this case, he is either studying or with his friends.
>
> Case 2: Suppose he is on campus. We know that if he is on campus, then he is with his friends. So, in this case, he is with his friends. Therefore, in this case, he is either studying or with his friends.

Either way, George is either studying or with his friends.

The natural deduction proof looks as follows:

$$\dfrac{A \vee B \qquad \dfrac{\dfrac{A \to C \qquad \overline{A}^{\,1}}{C}}{C \vee D} \qquad \dfrac{\dfrac{B \to D \qquad \overline{B}^{\,1}}{D}}{C \vee D}}{C \vee D}\,1$$

You should think about how the structure of this proof reflects the informal case-based argument above it.

For another example, here is a proof of $A \wedge (B \vee C) \to (A \wedge B) \vee (A \wedge C)$:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{A \wedge (B \vee C)}{}^{2}}{B \vee C} \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{A \wedge (B \vee C)}{}^{2}}{A} \qquad \dfrac{}{B}{}^{1}}{A \wedge B}}{(A \wedge B) \vee (A \wedge C)} \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{A \wedge (B \vee C)}{}^{2}}{A} \qquad \dfrac{}{C}{}^{1}}{A \wedge C}}{(A \wedge B) \vee (A \wedge C)}}{(A \wedge B) \vee (A \wedge C)}{}^{1}}{(A \wedge (B \vee C)) \to ((A \wedge B) \vee (A \wedge C))}{}^{2}$$

3.5 Some Logical Identities

Two propositional formulas, A and B, are said to be *logically equivalent* if $A \leftrightarrow B$ is provable. Logical equivalences are similar to identities like $x + y = y + x$ that occur in algebra. In particular, one can show that if two formulas are equivalent, then one can substitute one for the other in any formula, and the results will also be equivalent. (Some proof systems take this to be a basic rule, and interactive theorem provers can accommodate it, but we will *not* take it to be a fundamental rule of natural deduction.)

For reference, the following list contains some commonly used propositional equivalences, along with some noteworthy formulas. Think about why, intuitively, these formulas should be true.

1. Commutativity of $\wedge$: $A \wedge B \leftrightarrow B \wedge A$
2. Commutativity of $\vee$: $A \vee B \leftrightarrow B \vee A$
3. Associativity of $\wedge$: $(A \wedge B) \wedge C \leftrightarrow A \wedge (B \wedge C)$
4. Associativity of $\vee$ $(A \vee B) \vee C \leftrightarrow A \vee (B \vee C)$
5. Distributivity of $\wedge$ over $\vee$: $A \wedge (B \vee C) \leftrightarrow (A \wedge B) \vee (A \wedge C)$
6. Distributivity of $\vee$ over $\wedge$: $A \vee (B \wedge C) \leftrightarrow (A \vee B) \wedge (A \vee C)$
7. $(A \to (B \to C)) \leftrightarrow (A \wedge B \to C)$.
8. $(A \to B) \to ((B \to C) \to (A \to C))$
9. $((A \vee B) \to C) \leftrightarrow (A \to C) \wedge (B \to C)$
10. $\neg(A \vee B) \leftrightarrow \neg A \wedge \neg B$
11. $\neg(A \wedge B) \leftrightarrow \neg A \vee \neg B$
12. $\neg(A \wedge \neg A)$
13. $\neg(A \to B) \leftrightarrow A \wedge \neg B$
14. $\neg A \to (A \to B)$
15. $(\neg A \vee B) \leftrightarrow (A \to B)$
16. $A \vee \bot \leftrightarrow A$
17. $A \wedge \bot \leftrightarrow \bot$
18. $A \vee \neg A$
19. $\neg(A \leftrightarrow \neg A)$
20. $(A \to B) \leftrightarrow (\neg B \to \neg A)$
21. $(A \to C \vee D) \to ((A \to C) \vee (A \to D))$

22. $(((A \to B) \to A) \to A)$

All of these can be derived in natural deduction using the fundamental rules listed in Section 3.1. But some of them require the use of the *reductio ad absurdum* rule, or proof by contradiction, which we have not yet discussed in detail. We will discuss the use of this rule, and other patterns of classical logic, in the Chapter 5.

3.6 Exercises

When constructing proofs in natural deduction, use *only* the list of rules given in Section 3.1.

1. Give a natural deduction proof of $\neg(A \wedge B) \to (A \to \neg B)$.
2. Give a natural deduction proof of $(A \to C) \wedge (B \to \neg C) \to \neg(A \wedge B)$.
3. Give a natural deduction proof of $(A \wedge B) \to ((A \to C) \to \neg(B \to \neg C))$.
4. Take another look at Exercise 3 in the last chapter. Using propositional variables A, B, and C for "Alan likes kangaroos," "Betty likes frogs" and "Carl likes hamsters," respectively, express the three hypotheses in the previous problem as symbolic formulas, and then derive a contradiction from them in natural deduction.
5. Give a natural deduction proof of $A \vee B \to B \vee A$.
6. Give a natural deduction proof of $\neg A \wedge \neg B \to \neg(A \vee B)$
7. Give a natural deduction proof of $\neg(A \wedge B)$ from $\neg A \vee \neg B$. (You do not need to use proof by contradiction.)
8. Give a natural deduction proof of $\neg(A \leftrightarrow \neg A)$.
9. Give a natural deduction proof of $(\neg A \leftrightarrow \neg B)$ from hypothesis $A \leftrightarrow B$.

CHAPTER

FOUR

PROPOSITIONAL LOGIC IN LEAN

In this chapter, you will learn how to write proofs in Lean. We will start with a purely mechanical translation that will enable you to represent any natural deduction proof in Lean. We will see, however, that such a style of writing proofs is not very intuitive, nor does it yield very readable proofs. It also does not scale well.

We will then consider some mechanisms that Lean offers that support a more forward-directed style of argumentation. Since these proofs look more like informal proofs but can be directly translated to natural deduction, they will help us understand the relationship between the two.

4.1 Expressions for Propositions and Proofs

At its core, Lean is what is known as a *type checker*. This means that we can write expressions and ask the system to check that they are well formed, and also ask the system to tell us what type of object they denote. Try this:

```
variables A B C : Prop

#check A ∧ ¬ B → C
```

In the online version of this text, you can press the "try it!" button to copy the example to the editor window, press the "play" button, and then hover over the markers on the left to read the messages.

In the example, we declare three variables ranging over propositions, and ask Lean to check the expression `A ∧ ¬ B → C`. The output of the `#check` command is `A ∧ ¬ B → C : Prop`, which asserts that `A ∧ ¬ B → C` is of type `Prop`. In Lean, every well-formed expression has a type.

The logical connectives are rendered in unicode. The following chart shows you how you can type these symbols in the editor, and also provides ascii equivalents, for the purists among you.

Unicode	Ascii	Emacs
	true	
	false	
¬	not	`\not`, `\neg`
∧	/	`\and`
∨	/	`\or`
→	->	`\to`, `\r`, `\implies`
↔	<->	`\iff`, `\lr`
∀	forall	`\all`
∃	exists	`\ex`
λ	fun	`\lam`, `\fun`
≠	~=	`\ne`

So far, we have only talked about the first seven items on the list. We will discuss the quantifiers, lambda, and equality later. Try typing some expressions and checking them on your own. You should try changing one of the variables in the example above to D, or inserting a nonsense symbol into the expression, and take a look at the error message that Lean returns.

In addition to declaring variables, if P is any expression of type Prop, we can declare the hypothesis that P is true:

```
variables A B : Prop
variable  h : A ∧ ¬ B

#check h
```

Formally, what is going on is that any proposition can be viewed as a type, namely, the type of proofs of that proposition. A hypothesis, or premise, is just a variable of that type. Building proofs is then a matter of writing down expressions of the write type. For example, if P is any expression of type A ∧ B, then and.left P is an expression of type A, and and.right P is an expression of type B. In other words, if P is a proof of A ∧ B, and and.left P is a name for the proof you get by applying the left elimination rule for and:

$$\dfrac{\begin{array}{c}\vdots\\ P\\ \vdots\\ A \wedge B\end{array}}{A}$$

Similarly, and.right P is the proof of B you get by applying the right elimination rule. So, continuing the example above, we can write

```
variables A B : Prop
variable h : A ∧ ¬ B

#check and.left h
#check and.right h
```

The two expressions represent, respectively, these two proofs:

$$\dfrac{\overline{A \wedge \neg B}^{\,h}}{A} \qquad \dfrac{\overline{A \wedge \neg B}^{\,h}}{\neg B}$$

Notice that in this way of representing natural deduction proofs, there are no "free floating" hypotheses. Every hypothesis has a label. In Lean, we will typically use expressions like h, h1, h2, … to label hypotheses, but you can use any identifier you want.

If h1 is a proof of A and h2 is a proof of B, then and.intro h1 h2 is a proof of A ∧ B. So we can continue the example above:

```
variables A B : Prop
variable h : A ∧ ¬ B

#check and.intro (and.right h) (and.left h)
```

This corresponds to the following proof:

$$\dfrac{\dfrac{\overline{A \wedge \neg B}^{\,h}}{\neg B} \qquad \dfrac{\overline{A \wedge \neg B}^{\,h}}{A}}{\neg B \wedge A}$$

What about implication? The elimination rule is easy: if P_1 is a proof of A → B and P_2 is a proof of A then P_1 P_2 is a proof of B. Notice that we do not even need to name the rule: you just write P_1 followed by P_2, as though you are applying the first to the second. If P_1 and P_2 are compound expressions, put parentheses around them to make it clear where each one begins and ends.

```
variables A B C D : Prop

variable h1 : A → (B → C)
variable h2 : D → A
variable h3 : D
variable h4 : B

#check h2 h3
#check h1 (h2 h3)
#check (h1 (h2 h3)) h4
```

Lean adopts the convention that applications associate to the left, so that an expression `h1 h2 h3` is interpreted as `(h1 h2) h3`. Implications associate to the *right*, so that `A → B → C` is interpreted as `A → (B → C)`. This may seem funny, but it is a convenient way to represent implications that take multiple hypotheses, since an expression `A → B → C → D → E` means that `E` follows from `A`, `B`, `C`, and `D`. So the example above could be written as follows:

```
variables A B C D : Prop

variable h1 : A → B → C
variable h2 : D → A
variable h3 : D
variable h4 : B

#check h2 h3
#check h1 (h2 h3)
#check h1 (h2 h3) h4
```

Notice that parentheses are still needed in the expression `h1 (h2 h3)`.

The implication introduction rule is the tricky one, because it can cancel a hypothesis. In terms of Lean expressions, the rule translates as follows. Suppose `A` and `B` have type `Prop`, and, assuming `h` is the premise that `A` holds, `P` is proof of `B`, possibly involving `h`. Then the expression `assume h : A, P` is a proof of `A → B`. For example, we can construct a proof of `A → A ∧ A` as follows:

```
variable A : Prop

#check (assume h : A, and.intro h h)
```

Notice that we no longer have to declare `A` as a premise. The word `assume` makes the premise local to the expression in parentheses, and after the assumption is made, we can refer to `h`. Given the assumption `h : A`, `and.intro h h` is a proof of `A ∧ A`, and so the expression `assume h : A, and.intro h h` is a proof of `A → A ∧ A`. In this case, we could leave out the parentheses because the expression is unambiguous:

```
variable A : Prop

#check assume h : A, and.intro h h
```

Above, we proved `¬ B ∧ A` from the premise `A ∧ ¬ B`. We can instead obtain a proof of `A ∧ ¬ B → ¬ B ∧ A` as follows:

```
variables A B : Prop
#check (assume h : A ∧ ¬ B, and.intro (and.right h) (and.left h))
```

All we did was move the premise into a local `assume`.

(By the way, the `assume` command is just alternative syntax for the lambda symbol, so we could also have written this:

```
variables A B : Prop
#check (λ h : A ∧ ¬ B, and.intro (and.right h) (and.left h))
```

You will learn more about the lambda symbol later.)

4.2 More commands

Let us introduce a new Lean command, `example`. This command tells Lean that you are about to prove a theorem, or, more generally, write down an expression of the given type. It should then be followed by the proof or expression itself.

```
variables A B : Prop

example : A ∧ ¬ B → ¬ B ∧ A :=
assume h : A ∧ ¬ B,
and.intro (and.right h) (and.left h)
```

When given this command, Lean checks the expression after the `:=` and makes sure it has the right type. If so, it accepts the expression as a valid proof. If not, it raises and error.

Because the `example` command provides information as to the type of the expression that follows (in this case, the proposition being proved), it sometimes enables us to omit other information. For example, we can leave off the type of the assumption:

```
variables A B : Prop

example : A ∧ ¬ B → ¬ B ∧ A :=
assume h,
and.intro (and.right h) (and.left h)
```

Because Lean knows we are trying to prove an implication with premise `A ∧ ¬ B`, it can infer that when we write `assume h`, the identifier `h` labels the assumption `A ∧ ¬ B`.

We can also go in the other direction, and provide the system with *more* information, with the word `show`. If `A` is a proposition and `P` is a proof, the expression "`show A, from P`" means the same thing as `P` alone, but it signals the intention that `P` is a proof of `A`. When Lean checks this expression, it confirms that `P` really is a proof of `A`, before parsing the expression surrounding it. So, in our example, we could also write:

```
variables A B : Prop

example : A ∧ ¬ B → ¬ B ∧ A :=
assume h : A ∧ ¬ B,
show ¬ B ∧ A, from and.intro (and.right h) (and.left h)
```

We could even annotate the smaller expressions `and.right h` and `and.left h`, as follows:

```
variables A B : Prop

example : A ∧ ¬ B → ¬ B ∧ A :=
assume h : A ∧ ¬ B,
show ¬ B ∧ A, from and.intro
  (show ¬ B, from and.right h)
  (show A, from and.left h)
```

This is a good place to mention that Lean generally ignores whitespace, like indentation and returns. We could have written the entire example on a single line. In general, we will adopt conventions for line breaks and indentation that shows the structure of a proof and makes it easier to read.

Although in the examples above the `show` commands were not necessary, there are a number of good reasons to use it. First, and perhaps most importantly, it makes the proofs easier for us humans to read. Second, it makes the proofs easier to *write*: if you make a mistake in a proof, it is easier for Lean to figure out where you went wrong and provide a meaningful error message if you make your intentions clear. Finally, proving information in the `show` clause often makes it possible for you to omit information in other places, since Lean can infer that information from your stated intentions.

There are notational variants. Rather than declare variables and premises beforehand, you can also present them as "arguments" to the example, followed by a colon:

```
example (A B : Prop) : A ∧ ¬ B → ¬ B ∧ A :=
assume h : A ∧ ¬ B,
show ¬ B ∧ A, from and.intro (and.right h) (and.left h)
```

There are two more tricks that can help you write proofs in Lean. The first is using `sorry`, which is a magical term in Lean which provides a proof of anything at all. It is also known as "cheating." But cheating can help you construct legitimate proofs incrementally: if Lean accepts a proof with `sorry`'s, you know that you are on the right track so far. All you need to do is replace each `sorry` with a real proof to finish the task.

```
variables A B : Prop

example : A ∧ ¬ B → ¬ B ∧ A :=
assume h, sorry

example : A ∧ ¬ B → ¬ B ∧ A :=
assume h, and.intro sorry sorry

example : A ∧ ¬ B → ¬ B ∧ A :=
assume h, and.intro (and.right h) sorry

example : A ∧ ¬ B → ¬ B ∧ A :=
assume h, and.intro (and.right h) (and.left h)
```

The second trick is the use of *placeholders*, represented by the underscore symbol. When you write an underscore in an expression, you are asking the system to try to fill in the value for you. This falls short of calling full-blown automation to prove a theorem; rather, you are asking Lean to infer the value from the context. If you use an underscore where a proof should be, Lean typically will *not* fill in the proof, but it will give you an error message that tells you what is missing. This will help you write proof terms incrementally, in a backward-driven fashion. In the example above, try replacing each `sorry` by an underscore, `_`, and take a look at the resulting error messages. In each case, the error tells you what needs to be filled in, and the variables and hypotheses that are available to you at that stage.

One more tip: if you want to delimit the scope of variables or premises introduced with the `variables` command, put them in a block that begins with the word `section` and ends with the word `end`. We will use this mechanism below.

4.3 Building Natural Deduction Proofs

In this section, we describe a mechanical translation from natural deduction proofs, by giving a translation for each natural deduction rule. We have already seen some of the correspondences, but we repeat them all here, for completeness.

4.3.1 Implication

We have already explained that implication introduction is implemented with `assume`, and implication elimination is written as application.

```
variables A B : Prop

example : A → B :=
assume h : A,
show B, from sorry

section
  variable h1 : A → B
  variable h2 : A

  example : B := h1 h2
end
```

Since every example begins by declaring the necessary propositional variables, we will henceforth suppress that declaration in the text.

4.3.2 Conjunction

We have already seen that and introduction is implemented with `and.intro`, and the elimination rules are `and.left` and `and.right`.

```
section
  variables (h1 : A) (h2 : B)

  example : A ∧ B := and.intro h1 h2
end

section
  variable h : A ∧ B

  example : A := and.left h
  example : B := and.right h
end
```

4.3.3 Disjunction

The or introduction rules are given by `or.inl` and `or.inr`.

```
section
  variable h : A

  example : A ∨ B := or.inl h
end

section
  variable h : B

  example : A ∨ B := or.inr h
end
```

The elimination rule is the tricky one. To prove `C` from `A ∨ B`, you need three arguments: a proof `h` of `A ∨ B`, a proof of `C` from `A`, and a proof of `C` from `B`. Using line breaks and indentation to highlight the structure as a proof by cases, we can write it with the following form:

```
section
  variable h : A ∨ B

  example : C :=
  or.elim h
    (assume h1 : A,
      show C, from sorry)
    (assume h1 : B,
      show C, from sorry)
end
```

Notice that we can reuse the label `h1` in each branch, since, conceptually, the two branches are disjoint.

4.3.4 Negation

Internally, negation `¬ A` is defined by `A → false`, which you can think of as saying that `A` implies something impossible. The rules for negation are therefore similar to the rules for implication. To prove `¬ A`, assuming `A` and derive a contradiction.

```
section
  example : ¬ A :=
  assume h : A,
  show false, from sorry
end
```

If you have proved a negation `¬ A`, you can get a contradiction by applying it to a proof of `A`.

```
section
  variable h1 : ¬ A
  variable h2 : A

  example : false := h1 h2
end
```

4.3.5 Truth and falsity

The *ex falso* rule is called `false.elim`:

```
section
  variable h : false

  example : A := false.elim h
end
```

There isn't much to say about `true` beyond the fact that it is trivially true:

```
example : true := trivial
```

4.3.6 Bi-implication

The introduction rule for “if and only if” is `iff.intro`.

```
example : A ↔ B :=
iff.intro
  (assume h : A,
    show B, from sorry)
  (assume h : B,
    show A, from sorry)
```

As usual, we have chosen indentation to make the structure clear. Notice that the same label, `h`, can be used on both branches, with a different meaning in each, because the scope of an `assume` is limited to the expression in which it appears.

The elimination rules are `iff.elim_left` and `iff.elim_right`:

```
section
  variable h1 : A ↔ B
  variable h2 : A

  example : B := iff.elim_left h1 h2
end

section
  variable h1 : A ↔ B
  variable h2 : B

  example : A := iff.elim_right h1 h2
end
```

Lean recognizes the abbreviation `iff.mp` for `iff.and_elim_left`, where “mp” stands for “modus ponens”. Similarly, you can use `iff.mpr`, for “modus ponens reverse”, instead of `iff.and_elim_right`.

4.3.7 Reductio ad absurdum (proof by contradiction)

Finally, there is the rule for proof by contradiction, which we will discuss in greater detail in Chapter 5. It is included for completeness here.

The rule is called `by_contradiction`. It has one argument, which is a proof of `false` from `¬ A`. To use the rule, you have to ask Lean to allow classical reasoning, by writing `open classical`. You can do this at the beginning of the file, or any time before using it. It you say `open classical` in a section, it will remain in scope for that section.

```
section
  open classical

  example : A :=
  by_contradiction
    (assume h : ¬ A,
      show false, from sorry)
end
```

4.3.8 Examples

In the last chapter, we constructed the following proof $A \to C$ from $A \to B$ and $B \to C$:

$$\dfrac{\dfrac{\dfrac{\dfrac{1}{A} \qquad A \to B}{B} \qquad B \to C}{C}}{A \to C}\,1$$

We can model this in Lean as follows:

```
variables A B C : Prop

variable h1 : A → B
variable h2 : B → C

example : A → C :=
assume h : A,
show C, from h2 (h1 h)
```

Notice that we simply declare the uncanceled hypotheses as variables.

We also constructed the following proof:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{A \to (B \to C)}\,2 \qquad \dfrac{\dfrac{}{A \wedge B}\,1}{A}}{B \to C} \qquad \dfrac{\dfrac{}{A \wedge B}\,1}{B}}{\dfrac{C}{A \wedge B \to C}\,1}}{(A \to (B \to C)) \to (A \wedge B \to C)}\,2$$

Here is how it is written in Lean:

```
example (A B C : Prop) : (A → (B → C)) → (A ∧ B → C) :=
assume h1 : A → (B → C),
assume h2 : A ∧ B,
show C, from h1 (and.left h2) (and.right h2)
```

This works because `and.left h2` is a proof of `A`, and `and.right h2` is a proof of `B`.

Finally, we constructed the following proof of $A \wedge (B \vee C) \to (A \wedge B) \vee (A \wedge C)$:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{A \wedge (B \vee C)}\,2}{B \vee C} \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{A \wedge (B \vee C)}\,2}{A} \qquad \dfrac{}{B}\,1}{A \wedge B}}{(A \wedge B) \vee (A \wedge C)} \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{A \wedge (B \vee C)}\,2}{A} \qquad \dfrac{}{C}\,1}{A \wedge C}}{(A \wedge B) \vee (A \wedge C)}}{(A \wedge B) \vee (A \wedge C)}\,1}{(A \wedge (B \vee C)) \to ((A \wedge B) \vee (A \wedge C))}\,2$$

Here is a version in Lean:

```
example (A B C : Prop) : A ∧ (B ∨ C) → (A ∧ B) ∨ (A ∧ C) :=
assume h1 : A ∧ (B ∨ C),
or.elim (and.right h1)
  (assume h2 : B,
    show (A ∧ B) ∨ (A ∧ C),
      from or.inl (and.intro (and.left h1) h2))
  (assume h2 : C,
    show (A ∧ B) ∨ (A ∧ C),
      from or.inr (and.intro (and.left h1) h2))
```

In fact, bearing in mind that `assume` is alternative syntax for the symbol λ, and that Lean can often infer the type of an assumption, we can make the proof remarkably brief:

```
example (A B C : Prop) : A ∧ (B ∨ C) → (A ∧ B) ∨ (A ∧ C) :=
λ h1, or.elim (and.right h1)
  (λ h2, or.inl (and.intro (and.left h1) h2))
  (λ h2, or.inr (and.intro (and.left h1) h2))
```

The proof is cryptic, though. Using such a style makes proofs hard to write, read, understand, maintain, and debug. In the next section we will describe a remarkably simple device that makes it much easier to understand what is going on.

4.4 Forward Reasoning

Lean supports forward reasoning by allowing you to write proofs using the `have` command.

```
variables A B C : Prop

variable h1 : A → B
variable h2 : B → C

example : A → C :=
assume h : A,
have h3 : B, from h1 h,
show C, from h2 h3
```

Writing a proof with `have h : A, from P, ... h ...` has the same effect as writing `... P ...`. This `have` command checks that `P` is a proof of `A`, and then give you the label `h` to use in place of `P`. Thus the last line of the previous proof can be thought of as abbreviating `show C, from h2 (h1 h)`, since `h3` abbreviates `h1 h`. Such abbreviations can make a big difference, especially when the proof `P` is very long.

There are a number of advantages to using `have`. For one thing, it makes the proof more readable: the example above states `B` explicitly as an auxiliary goal. It can also save repetition: `h3` can be used repeatedly after it is introduced, without duplicating the proof. Finally, it makes it easier to construct and debug the proof: stating `B` as an auxiliary goal makes it easier for Lean to deliver an informative error message when the goal is not properly met.

In the last section, we considered the following proof:

```
example (A B C : Prop) : (A → (B → C)) → (A ∧ B → C) :=
assume h1 : A → (B → C),
assume h2 : A ∧ B,
show C, from h1 (and.left h2) (and.right h2)
```

Using `have`, it can be written more perspicuously as follows:

```
example (A B C : Prop) : (A → (B → C)) → (A ∧ B → C) :=
assume h1 : A → (B → C),
assume h2 : A ∧ B,
have h3 : A, from and.left h2,
have h4 : B, from and.right h2,
show C, from h1 h3 h4
```

We can be even more verbose, and add another line:

```
example (A B C : Prop) : (A → (B → C)) → (A ∧ B → C) :=
assume h1 : A → (B → C),
assume h2 : A ∧ B,
have h3 : A, from and.left h2,
have h4 : B, from and.right h2,
have h5 : B → C, from h1 h3,
show C, from h5 h4
```

Adding more information doesn't always make a proof more readable; when the individual expressions are small and easy enough to understand, spelling them out in detail can introduce clutter. As you learn to use Lean, you will have to develop your own style, and use your judgment to decide which steps to make explicit.

Here is how some of the basic inferences look, when expanded with `have`. In the and-introduction rule, it is a matter showing each conjunct first, and then putting them together:

```
example (A B : Prop) : A ∧ B → B ∧ A :=
assume h1 : A ∧ B,
have h2 : A, from and.left h1,
have h3 : B, from and.right h1,
show B ∧ A, from and.intro h3 h2
```

Compare that with this version, which instead states first that we will use the `and.intro` rule, and then makes the two resulting goals explicit:

```
example (A B : Prop) : A ∧ B → B ∧ A :=
assume h1 : A ∧ B,
show B ∧ A, from
  and.intro
    (show B, from and.right h1)
    (show A, from and.left h1)
```

Once again, at issue is only readability. Lean does just fine with the following short version:

```
example (A B : Prop) : A ∧ B → B ∧ A :=
λ h, and.intro (and.right h) (and.left h)
```

When using the or-elimination rule, it is often clearest to state the relevant disjunction explicitly:

```
example (A B C : Prop) : C :=
have h : A ∨ B, from sorry,
show C, from or.elim h
  (assume h1 : A,
    show C, from sorry)
  (assume h2 : B,
    show C, from sorry)
```

Here is a `have`-structured presentation of an example from the previous section:

```
example (A B C : Prop) : A ∧ (B ∨ C) → (A ∧ B) ∨ (A ∧ C) :=
assume h1 : A ∧ (B ∨ C),
have h2 : A, from and.left h1,
have h3 : B ∨ C, from and.right h1,
show (A ∧ B) ∨ (A ∧ C), from
  or.elim h3
    (assume h4 : B,
      have h5 : A ∧ B, from and.intro h2 h4,
      show (A ∧ B) ∨ (A ∧ C), from or.inl h5)
    (assume h4 : C,
      have h5 : A ∧ C, from and.intro h2 h4,
      show (A ∧ B) ∨ (A ∧ C), from or.inr h5)
```

4.5 Definitions and Theorems

Lean allows us to name definitions and theorems for later use. For example, here is a definition of a new "connective":

```
def triple_and (A B C : Prop) : Prop :=
A ∧ (B ∧ C)
```

As with the example command, it does not matter whether the arguments A, B, and C are declared beforehand with the variables command, or with the definition itself. We can then apply the definition to any expressions:

```
variables D E F G : Prop

#check triple_and (D ∨ E) (¬ F → G) (¬ D)
```

Later, we will see more interesting examples of definitions, like the following function from natural numbers to natural numbers, which doubles its input:

```
def double (n : ℕ) : ℕ := n + n
```

What is more interesting right now is that Lean also allows us to name theorems, and use them later, as rules of inference. For example, consider the following theorem:

```
theorem and_commute (A B : Prop) : A ∧ B → B ∧ A :=
assume h, and.intro (and.right h) (and.left h)
```

Once we have defined it, we can use it freely:

```
variables C D E : Prop
variable h1 : C ∧ ¬ D
variable h2 : ¬ D ∧ C → E

example : E := h2 (and_commute C (¬ D) h1)
```

It is annoying in this example that we have to give the arguments C and ¬ D explicitly, because they are implicit in h1. In fact, Lean allows us to tell this to Lean in the definition of and_commute:

```
theorem and_commute {A B : Prop} : A ∧ B → B ∧ A :=
assume h, and.intro (and.right h) (and.left h)
```

here the squiggly braces indicate that the arguments A and B are *implicit*, which is to say, Lean should infer them from the context when the theorem is used. We can then write the following instead:

```
variables C D E : Prop
variable h1 : C ∧ ¬ D
variable h2 : ¬ D ∧ C → E

example : E := h2 (and_commute h1)
```

Indeed, Lean's library has a theorem, and_comm, defined in exactly this way.

By the way, we could avoid the assume step in the proof of and_comm by making the hypothesis into an argument:

```
theorem and_commute {A B : Prop} (h : A ∧ B) : B ∧ A :=
and.intro (and.right h) (and.left h)
```

The two definitions yield the same result.

Definitions and theorems are important in mathematics; they allow us to build up complex theories from fundamental principles. Lean also accepts the word `lemma` instead of `theorem`.

What is interesting is that in interactive theorem proving, we can even define familiar patterns of inference. For example, all of the following inferences were mentioned in the last chapter:

```
namespace hide

variables {A B : Prop}

theorem or_resolve_left (h1 : A ∨ B) (h2 : ¬ A) : B :=
or.elim h1
  (assume h3 : A, show B, from false.elim (h2 h3))
  (assume h3 : B, show B, from h3)

theorem or_resolve_right (h1 : A ∨ B) (h2 : ¬ B) : A :=
or.elim h1
  (assume h3 : A, show A, from h3)
  (assume h3 : B, show A, from false.elim (h2 h3))

theorem absurd (h1 : ¬ A) (h2 : A) : B :=
false.elim (h1 h2)

end hide
```

In fact, Lean's library defines `or.resolve_left`, `or.resolve_right`, and `absurd`. We used the `namespace` command to avoid naming conflicts, which would have raised an error.

When we ask you to prove basic facts from propositional logic in Lean, as with propositional logic, our goal is to have you learn how to use Lean's primitives. As a result, for those exercises, you should not use facts from the library. As we move towards real mathematics, however, you can use facts from the library more freely.

4.6 Additional Syntax

In this section, we describe some extra syntactic features of Lean, for power users. The syntactic gadgets are often convenient, and sometimes make proofs look prettier.

For one thing, you can use subscripted numbers with a backslash. For example, you can write h_1 by typing `h\1`. The labels are irrelevant to Lean, so the difference is only cosmetic.

Another feature is that you can omit the label in an `assume` statement, providing an "anonymous" hypothesis. You can then refer back to the last anonymous assumption using the keyword `this`:

```
example : A → A ∨ B :=
assume : A,
show A ∨ B, from or.inl this
```

Alternatively, you can refer back to unlabeled assumptions by putting them in French quotes:

```
example : A → B → A ∧ B :=
assume : A,
assume : B,
show A ∧ B, from and.intro ‹A› ‹B›
```

You can also use the word `have` without giving a label, and refer back to them using the same conventions. Here is an example that uses these features:

```
theorem my_theorem {A B C : Prop} :
  A ∧ (B ∨ C) → (A ∧ B) ∨ (A ∧ C) :=
assume h : A ∧ (B ∨ C),
have A, from and.left h,
have B ∨ C, from and.right h,
show (A ∧ B) ∨ (A ∧ C), from
  or.elim ‹B ∨ C›
    (assume : B,
      have A ∧ B, from and.intro ‹A› ‹B›,
      show (A ∧ B) ∨ (A ∧ C), from or.inl this)
    (assume : C,
      have A ∧ C, from and.intro ‹A› ‹C›,
      show (A ∧ B) ∨ (A ∧ C), from or.inr this)
```

Another trick is that you can write `h.left` and `h.right` instead of `and.left h` and `and.right h` whenever `h` is a conjunction, and you can write `⟨h1, h2⟩` instead of `and.intro h1 h2` whenever Lean can figure out that a conjunction is what you are trying to prove. With these conventions, you can write the following:

```
example (A B : Prop) : A ∧ B → B ∧ A :=
assume h : A ∧ B,
show B ∧ A, from ⟨h.right, h.left⟩
```

This is nothing more than shorthand for the following:

```
example (A B : Prop) : A ∧ B → B ∧ A :=
assume h : A ∧ B,
show B ∧ A, from and.intro (and.right h) (and.left h)
```

Even more concisely, you can write this:

```
example (A B : Prop) : A ∧ B → B ∧ A :=
assume h, ⟨h.right, h.left⟩
```

You can even take apart a conjunction with an `assume`, so that this works:

```
example (A B : Prop) : A ∧ B → B ∧ A :=
assume ⟨h₁, h₂⟩, ⟨h₂, h₁⟩
```

Similarly, if `h` is a biconditional, you can write `h.mp` and `h.mpr` instead of `iff.mp h` and `iff.mpr h`, and you can write `⟨h1, h2⟩` instead of `iff.intro h1 h2`. As a result, Lean understands these proofs:

```
example (A B : Prop) : B ∧ (A ↔ B) → A :=
assume ⟨hB, hAB⟩,
hAB.mpr hB

example (A B : Prop) : A ∧ B ↔ B ∧ A :=
⟨assume ⟨h₁, h₂⟩, ⟨h₂, h₁⟩, assume ⟨h₁, h₂⟩, ⟨h₂, h₁⟩⟩
```

Finally, you can add comments to your proofs in two ways. First, any text after a double-dash `--` until the end of a line is ignored by the Lean processor. Second, any text between `/-` and `-/` denotes a block comment, and is also ignored. You can nest block comments.

```
/- This is a block comment.
   It can fill multiple lines. -/
```

```
example (A : Prop) : A → A :=
assume : A,         -- assume the antecedent
show A, from this   -- use it to establish the conclusion
```

4.7 Exercises

Prove the following in Lean:

```
variables A B C D : Prop

example : A ∧ (A → B) → B :=
sorry

example : A → ¬ (¬ A ∧ B) :=
sorry

example : ¬ (A ∧ B) → (A → ¬ B) :=
sorry

example (h₁ : A ∨ B) (h₂ : A → C) (h₃ : B → D) : C ∨ D :=
sorry

example (h : ¬ A ∧ ¬ B) : ¬ (A ∨ B) :=
sorry

example : ¬ (A ↔ ¬ A) :=
sorry
```

CHAPTER

FIVE

CLASSICAL REASONING

If we take all the rules of propositional logic we have seen so far and exclude *reductio ad absurdum*, or proof by contradiction, we have what is known as *intuitionistic logic*. In intuitionistic logic, it is possible to view proofs in computational terms: a proof of $A \wedge B$ is a proof of A paired with a proof of B, a proof of $A \to B$ is a procedure which transforms evidence for A into evidence for B, and a proof of $A \vee B$ is a proof of one or the other, tagged so that we know which is the case. The *ex falso* rule makes sense only because we expect that there is no proof of falsity; it is like the empty data type.

Proof by contradiction does not fit it well with this world view: from a proof of a contradiction from $\neg A$, we are supposed to magically produce a proof of A. We will see that with proof by contradiction, we can prove the law of the excluded middle, $A \vee \neg A$. From a computational perspective, this would say that we can ways decide whether or not A is true.

Classical reasoning does introduce a number of principles into logic, however, that can be used to simplify reasoning. In this chapter, we will consider these principles, and see how they follow from the basic rules.

5.1 Proof by Contradiction

Remember that in natural deduction, proof by contradiction is expressed by the following pattern:

$$\cfrac{\cfrac{\overline{\neg A}^{\,1}}{\vdots}}{\cfrac{\bot}{A}\,1}$$

The assumption $\neg A$ is canceled at the final inference.

In Lean, the inference is named `by_contradiction`, and since it is a classical rule, we have to use the command `open classical` before it is available. Once we do so, the pattern of inference is expressed as follows:

```
open classical

variable (A : Prop)

example : A :=
by_contradiction
  (assume h : ¬ A,
    show false, from sorry)
```

One of them most important consequences of this rule is the law of the excluded middle. In mathematical arguments, one often splits a proof into two cases, assuming first A and then $\neg A$. Using the elimination rule for disjunction, this is equivalent to using $A \vee \neg A$, a classical principle known as the law of the excluded middle. Here is a proof of this, in natural deduction, using a proof by contradiction:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\neg(A \vee \neg A)}{}^{2} \qquad \dfrac{\dfrac{}{A}{}^{1}}{A \vee \neg A}}{\bot}}{\neg A}{}^{1}}{A \vee \neg A} \qquad \dfrac{}{\neg(A \vee \neg A)}{}^{1}}{\bot}}{A \vee \neg A}{}^{1}$$

Here is the same proof rendered in Lean:

```
open classical

variable (A : Prop)

example : A ∨ ¬ A :=
by_contradiction
  (assume h1 : ¬ (A ∨ ¬ A),
    have h2 : ¬ A, from
      assume h3 : A,
      have h4 : A ∨ ¬ A, from or.inl h3,
      show false, from h1 h4,
    have h5 : A ∨ ¬ A, from or.inr h2,
    show false, from h1 h5)
```

The principle is known as the law of the excluded middle because it says that a proposition `A` is either true or false; there is no middle ground. As a result, the theorem is named `em` in the Lean library. For any proposition `A`, `em A` denotes a proof of `A ∨ ¬ A`, and you are free to use it any time `classical` is open:

```
open classical

example (A : Prop) : A ∨ ¬ A :=
or.elim (em A)
  (assume : A, or.inl this)
  (assume : ¬ A, or.inr this)
```

Or even more simply:

```
open classical

example (A : Prop) : A ∨ ¬ A :=
em A
```

In fact, we can go in the other direction, and use the law of the excluded middle to justify proof by contradiction. You are asked to do this in the exercises.

Proof by contradiction is also equivalent to the principle $\neg\neg A \leftrightarrow A$. The implication from right to left holds intuitionistically; the other implication is classical, and is known as *double-negation elimination*. Here is a proof in natural deduction:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\neg\neg A}{}^{2} \qquad \dfrac{}{\neg A}{}^{1}}{\bot}}{A}{}^{1} \qquad \dfrac{\dfrac{\dfrac{}{\neg A}{}^{1} \qquad \dfrac{}{A}{}^{2}}{\bot}}{\neg\neg A}{}^{1}}{\neg\neg A \leftrightarrow A}{}^{2}$$

And here is the corresponding proof in Lean:

```
open classical

example (A : Prop) : ¬ ¬ A ↔ A :=
iff.intro
  (assume h1 : ¬ ¬ A,
    show A, from by_contradiction
      (assume h2 : ¬ A,
        show false, from h1 h2))
  (assume h1 : A,
    show ¬ ¬ A, from assume h2 : ¬ A, h2 h1)
```

In the next section, we will derive a number of classical rules and equivalences. These are tricky to prove. In general, to use classical reasoning in natural deduction, we need to extend the general heuristic presented in Section 3.3 as follows:

1. First, work backwards from the conclusion, using the introduction rules.
2. When you have run out things to do in the first step, use elimination rules to work forwards.
3. If all else fails, use a proof by contradiction.

Sometimes a proof by contradiction is necessary, but when it isn't, it can be less informative by a direct proof. Suppose, for example, we want to prove $A \wedge B \wedge C \to D$. In a direct proof, we assume A, B, and C, and work towards D. Along the way, we will derive other consequences of A, B, and C, and these may be useful in other contexts. If we use proof by contradition, on the other hand, we assume A, B, C, and $\neg D$, and try to prove $\bot$. In that case, we are working in an inconsistent context; any auxiliary results we may obtain that way are subsumed by the fact that we ultimately $\bot$ is a consequence of the hypotheses.

5.2 Some Classical Principles

We have already seen that $A \vee \neg A$ and $\neg\neg A \leftrightarrow A$ are two important theorems of classical propositional logic. In this section we will provide some more theorems, rules, and equivalences. Some will be proved here, but most will be left to you in the exercises. In ordinary mathematics, these are generally used without comment. It is nice to know, however, that they can all be justified using the basic rules of classical natural deduction.

If $A \to B$ is any implication, the assertion $\neg B \to \neg A$ is known as the *contrapositive*. Every implication implies its contrapositive, and the other direction is true classically:

$$\dfrac{\dfrac{\dfrac{\dfrac{\neg B \to \neg A \qquad \dfrac{}{\neg B}{\scriptstyle 1}}{\neg A} \qquad \dfrac{}{A}{\scriptstyle 2}}{\bot}}{B}{\scriptstyle 1}}{A \to B}{\scriptstyle 2}$$

Here is another example. Intuitively, asserting "if A then B" is equivalent to saying that it cannot be the case that A is true and B is false. Classical reasoning is needed to get us from the second statement to the first.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\neg(A \wedge \neg B)}{\scriptstyle 3} \qquad \dfrac{\dfrac{}{A}{\scriptstyle 2} \qquad \dfrac{}{\neg B}{\scriptstyle 1}}{A \wedge \neg B}}{\bot}}{B}{\scriptstyle 1}}{A \to B}{\scriptstyle 2}}{\neg(A \wedge \neg B) \to (A \to B)}{\scriptstyle 3}$$

Here are the same proofs, rendered in Lean:

```
open classical

variables (A B : Prop)

example (h : ¬ B → ¬ A) : A → B :=
assume h1 : A,
show B, from
  by_contradiction
    (assume h2 : ¬ B,
      have h3 : ¬ A, from h h2,
      show false, from h3 h1)

example (h : ¬ (A ∧ ¬ B)) : A → B :=
assume : A,
show B, from
  by_contradiction
    (assume : ¬ B,
      have A ∧ ¬ B, from and.intro ‹A› this,
      show false, from h this)
```

Notice that in the second example, we used an anonymous `assume` and an anonymous `have`. We used the brackets `\f<` and `\f>` to write `‹A›`, referring back to the first assumption. The first use of the word `this` refers back to the assumption `¬ B`, while the second one refers back to the `have`.

Knowing that we can prove the law of the excluded middle, it is convenient to use it in classical proofs. Here is an example, with a proof of $(A \to B) \vee (B \to A)$:

$$\dfrac{\dfrac{}{B \vee \neg B} \qquad \dfrac{\dfrac{\dfrac{}{B}{\scriptstyle 1}}{A \to B}}{(A \to B) \vee (B \to A)} \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\neg B}{\scriptstyle 1} \quad \dfrac{}{B}{\scriptstyle 2}}{\bot}}{A}}{B \to A}{\scriptstyle 2}}{(A \to B) \vee (B \to A)}}{(A \to B) \vee (B \to A)}{\scriptstyle 1}$$

Here is the corresponding proof in Lean:

```
open classical

variables (A B : Prop)

example : (A → B) ∨ (B → A) :=
or.elim (em B)
  (assume h : B,
    have A → B, from
      assume : A,
      show B, from h,
    show (A → B) ∨ (B → A),
      from or.inl this)
  (assume h : ¬ B,
    have B → A, from
      assume : B,
      have false, from h this,
      show A, from false.elim this,
    show (A → B) ∨ (B → A),
      from or.inr this)
```

Using classical reasoning, implication can be rewritten in terms of disjunction and negation:

$$(A \to B) \leftrightarrow \neg A \vee B$$

The forward direction requires classical reasoning.

The following equivalences are known as De Morgan's laws:

$$\begin{aligned}\neg(A \vee B) &\leftrightarrow \neg A \wedge \neg B\\ \neg(A \wedge B) &\leftrightarrow \neg A \vee \neg B\end{aligned}$$

The forward direction of the second of these requires classical reasoning.

Using these identities, we can always push negations down to propositional variables. For example, we have

$$\begin{aligned}\neg(\neg A \wedge B \to C) &\leftrightarrow \neg(\neg(\neg A \wedge B) \vee C)\\ &\leftrightarrow \neg\neg(\neg A \wedge B) \wedge \neg C\\ &\leftrightarrow \neg A \wedge B \wedge \neg C\end{aligned}$$

A formula built up from $\wedge$, $\vee$, and $\neg$ in which negations only occur at variables is said to be in *negation normal form*.

In fact, using distributivity laws, one can go on to ensure that all the disjunctions are on the outside, so that the formulas is a big or of and's of propositional variables and negated propositional variables. Such a formula is said to be in *disjunctive normal form*. Alternatively, all the and's can be brought to the outside. Such a formula is said to be in *conjunctive normal form*. An exercise below, however, shows that putting formulas in disjunctive or conjunctive normal form can make them much longer.

5.3 Exercises

1. Show how to derive the proof-by-contradiction rule from the law of the excluded middle, using the other rules of natural deduction. In other words, assume you have a proof of $\bot$ from $\neg A$. Using $A \vee \neg A$ as a hypothesis, but *without* using the rule RAA, show how you can go on to derive A.
2. Give a natural deduction proof of $\neg(A \wedge B)$ from $\neg A \vee \neg B$. (You do not need to use proof by contradiction.)
3. Construct a natural deduction proof of $\neg A \vee \neg B$ from $\neg(A \wedge B)$. You can do it as follows:
   (a) First, prove $\neg B$, and hence $\neg A \vee \neg B$, from $\neg(A \wedge B)$ and A.
   (b) Use this to construct a proof of $\neg A$, and hence $\neg A \vee \neg B$, from $\neg(A \wedge B)$ and $\neg(\neg A \vee \neg B)$.
   (c) Use this to construct a proof of a contradiction from $\neg(A \wedge B)$ and $\neg(\neg A \vee \neg B)$.
   (d) Using proof by contradiction, this gives you a proof of $\neg A \vee \neg B$ from $\neg(A \wedge B)$.
4. Give a natural deduction proof of $\neg A \vee B$ from $A \to B$. You may use the law of the excluded middle.
5. Put $(A \vee B) \wedge (C \vee D) \wedge (E \vee F)$ in disjunctive normal form, that is, write it as a big "or" of "and"'s.
6. Prove `¬ (A ∧ B) → ¬ A ∨ ¬ B` by replacing the sorry's below by proofs.

```
open classical
variables {A B C : Prop}

-- Prove ¬ (A ∧ B) → ¬ A ∨ ¬ B by replacing the sorry's below
-- by proofs.

lemma step1 (h₁ : ¬ (A ∧ B)) (h₂ : A) : ¬ A ∨ ¬ B :=
have ¬ B, from sorry,
show ¬ A ∨ ¬ B, from or.inr this

lemma step2 (h₁ : ¬ (A ∧ B)) (h₂ : ¬ (¬ A ∨ ¬ B)) : false :=
have ¬ A, from
  assume : A,
  have ¬ A ∨ ¬ B, from step1 h₁ ‹A›,
  show false, from h₂ this,
show false, from sorry

theorem step3 (h : ¬ (A ∧ B)) : ¬ A ∨ ¬ B :=
by_contradiction
  (assume h' : ¬ (¬ A ∨ ¬ B),
    show false, from step2 h h')
```

7. Also do these:

```
open classical
variables {A B C : Prop}

example (h : ¬ B → ¬ A) : A → B :=
sorry

example (h : A → B) : ¬ A ∨ B :=
sorry
```

CHAPTER

SIX

SEMANTICS OF PROPOSITIONAL LOGIC

Classically, we think of propositional variables as ranging over statements that can be true or false. And, intuitively, we think of a proof system as telling us what propositional formulas *have to* be true, no matter what the variables stand for. For example, the fact that we can prove C from the hypotheses A, B, and $A \wedge B \to C$ seems to tell us that whenever the hypotheses are true, then C has to be true as well.

Making sense of this involves stepping outside the system and giving an account of truth — more precisely, the conditions under which a propositional formula is true. This is one of the things that symbolic logic was designed to do, and the task belongs to the realm of *semantics*. Formulas and formal proofs are *syntactic* notions, which is to say, they are represented by symbols and symbolic structures. Truth is a *semantic* notion, in that it ascribes a type of *meaning* to certain formulas.

Syntactically, we were able to ask and answer questions like the following:

- Given a set of hypotheses, Γ, and a formula, A, can we derive A from Γ?
- What formulas can be derived from Γ?
- What hypotheses are needed to derive A?

The questions we consider semantically are different:

- Given an assignment of truth values to the propositional variables occurring in a formula A, is A true or false?
- Is there any truth assignment that makes A true?
- Which are the truth assignments that make A true?

In this chapter, we will not provide a fully rigorous mathematical treatment of syntax and semantics. That subject matter is appropriate to a more advanced and focused course on mathematical logic. But we will discuss semantic issues in enough detail to give you a good sense of what it means to think semantically, as well as a sense of how to make pragmatic use of semantic notions.

6.1 Truth Values and Assignments

The first notion we will need is that of a *truth value*. We have already seen two, namely, “true” and “false.” We will use the symbols $\mathbf{T}$ and $\mathbf{F}$ to represent these in informal mathematics. These are the values that $\top$ and $\bot$ are intended to denote in natural deduction, and `true` and `false` are intended to denote in Lean.

In this text, we will adopt a “classical” notion of truth, following our discussion in Section 5. This can be understood in various ways, but, concretely, it comes down to this: we will assume that any proposition is either true or false (but, of course, not both). This conception of truth is what underlies the law of the excluded middle, $A \vee \neg A$. Semantically, we read this sentence as saying “either A is true, or $\neg A$ is true.” Since, in our semantic interpretation, $\neg A$ is true exactly when A is false, the law of the excluded middle says that A is either true or false.

The next notion we will need is that of a *truth assignment*, which is simply a function that assigns a truth value to each element of a propositional variables. In this section, we will distinguish between propositional variables and arbitrary formulas by using letters $P, Q, R, \ldots$ for the former and $A, B, C, \ldots$ for the latter. For example, the function v defined by

- $v(P) := \mathbf{T}$
- $v(Q) := \mathbf{F}$
- $v(R) := \mathbf{F}$
- $v(S) := \mathbf{T}$

is a truth assignment for the set of variables $\{P, Q, R, S\}$.

Intuitively, a truth assignment describes a possible "state of the world." Going back to the Malice and Alice puzzle, let's suppose the following letters are shorthand for the statements:

- $P :=$ Alice's brother was the victim
- $Q :=$ Alice was the killer
- $R :=$ Alice was in the bar

In the world described by the solution to the puzzle, the first and third statements are true, and the second is false. So our truth assignment gives the value $\mathbf{T}$ to P and R, and the value $\mathbf{F}$ to Q.

Once we have a truth assignment v to a set of propositional variables, we can extend it to a *valuation function* $\bar{v}$, which assigns a value of true or false to every propositional formula that depends only on these variables. The function $\bar{v}$ is defined recursively, which is to say, formulas are evaluated from the bottom up, so that value assigned to a compound formula is determined by the values assigned to its components. Formally, the function is defined as follows:

- $\bar{v}(\top) = \mathbf{T}$
- $\bar{v}(\bot) = \mathbf{F}$
- $\bar{v}(\ell) = v(\ell)$, where ℓ is any propositional variable.
- $\bar{v}(\neg A) = \mathbf{T}$ if $\bar{v}(A)$ is $\mathbf{F}$, and vice versa.
- $\bar{v}(A \wedge B) = \mathbf{T}$ if $\bar{v}(A)$ and $\bar{v}(B)$ are both $\mathbf{T}$, and $\mathbf{F}$ otherwise.
- $\bar{v}(A \vee B) = \mathbf{T}$ if at least one of $\bar{v}(A)$ and $\bar{v}(B)$ is $\mathbf{T}$; otherwise $\mathbf{F}$.
- $\bar{v}(A \to B) = \mathbf{T}$ if either $\bar{v}(B)$ is $\mathbf{T}$ or $\bar{v}(A)$ is $\mathbf{F}$, and $\mathbf{F}$ otherwise. (Equivalently, $\bar{v}(A \to B) = \mathbf{F}$ if $\bar{v}(A)$ is $\mathbf{T}$ and $\bar{v}(B)$ is $\mathbf{F}$, and $\mathbf{T}$ otherwise.)

The rules for conjunction and disjunction are easy to understand. "A and B" is true exactly when A and B are both true; "A or B" is true when at least one of A or B is true.

Understanding the rule for implication is trickier. People are often surprised to hear that any if-then statement with a false hypothesis is supposed to be true. The statement "if I have two heads, then circles are squares" may sound like it ought to be false, but by our reckoning, it comes out true. To make sense of this, think about the difference between the two sentences:

- "If I have two heads, then circles are squares."
- "If I had two heads, then circles would be squares."

The second sentence is an example of a *counterfactual* implication. It asserts something about how the world might change, if things were other than they actually are. Philosophers have studied counterfactuals for centuries, but mathematical logic is concerned with the first sentence, a *material* implication. The material implication asserts something about the way the world is right now, rather than the way it might have been. Since it is false that I have two heads, the statement "if I have two heads, then circles are squares" is true.

Why do we evaluate material implication in this way? Once again, let us consider the true sentence "every natural number that is prime and greater than two is odd." We can interpret this sentence as saying that all of the (infinitely many) sentences in this list are true:

- if 0 is prime and greater than 2, then 0 is odd
- if 1 is prime and greater than 2, then 1 is odd
- if 2 is prime and greater than 2, then 2 is odd
- if 3 is prime and greater than 2, then 3 is odd
- …

The first sentence on this list is a lot like our "two heads" example, since both the hypothesis and the conclusion are false. But since it is an instance of a statement that is true in general, we are committed to assigning it the value **T**. The second sentence is a different: the hypothesis is still false, but here the conclusion is true. Together, these tell us that whenever the hypothesis is false, the conditional statement should be true. The fourth sentence has a true hypothesis and a true conclusion. So from the second and fourth sentences, we see that whenever the conclusion is true, the conditional should be true as well. Finally, it seems clear that the sentence "if 3 is prime and greater than 2, then 3 is even" should *not* be true. This pattern, where the hypothesis is true and the conclusion is false, is the only one for which the conditional will be false.

Let us motivate the semantics for material implication another way, using the deductive rules described in the last chapter. Notice that, if B is true, we can prove $A \to B$ without any assumptions about A.

$$\frac{B}{A \to B}$$

This follows from the proper reading of the implication introduction rule: given B, one can always infer $A \to B$, and then cancel an assumption A, *if there is one*. If A was never used in the proof, the conclusion is simply weaker than it needs to be. This inference is validated in Lean:

```
variables A B : Prop
variable hB : B

example : A → B :=
assume hA : A,
  show B, from hB
```

Similarly, if A is false, we can prove $A \to B$ without any assumptions about B:

$$\frac{\dfrac{\neg A \qquad \overline{A}^{\,1}}{\bot}}{A \to B}\,1$$

In Lean:

```
variables A B : Prop
variable hnA : ¬ A

example : A → B :=
assume hA : A,
  show B, from false.elim (hnA hA)
```

Finally, if A is true and B is false, we can prove $\neg(A \to B)$:

$$\dfrac{\dfrac{\neg B \qquad \dfrac{\dfrac{}{A \to B}{}^{1} \qquad A}{B}}{\bot}}{\neg(A \to B)}{}^{1}$$

Once again, in Lean:

```
variables A B : Prop
variable hA : A
variable hnB : ¬B

example : ¬ (A → B) :=
assume h : A → B,
have hB : B, from h hA,
show false, from hnB hB
```

Now that we have defined the truth of any formula relative to a truth assignment, we can answer our first semantic question: given an assignment v of truth values to the propositional variables occurring in some formula φ, how do we determine whether or not φ is true? This amounts to evaluating $\bar{v}(\varphi)$, and the recursive definition of φ gives a recipe: we evaluate the expressions occurring in φ from the bottom up, starting with the propositional variables, and using the evaluation of an expression's components to evaluate the expression itself. For example, suppose our truth assignment v makes A and B true and C false. To evaluate $(B \to C) \vee (A \wedge B)$ under v, note that the expression $B \to C$ comes out false and the expression $A \wedge B$ comes out true. Since a disjunction "false or true" is true, the entire formula is true.

We can also go in the other direction: given a formula, we can attempt to find a truth assignment that will make it true (or false). In fact, we can use Lean to evaluate formulas for us. In the example that follows, you can assign any set of values to the proposition symbols `A`, `B`, `C`, `D`, and `E`. When you run Lean on this input, the output of the `eval` statement is the value of the expression.

```
-- Define your truth assignment here
def A := tt
def B := ff
def C := tt
def D := tt
def E := ff

def test (p : Prop) [decidable p] : string :=
if p then "true" else "false"

#eval test ((A ∧ B) ∨ ¬ C)
#eval test (A → D)
#eval test (C → (D ∨ ¬E))
#eval test (¬(A ∧ B ∧ C ∧ D))
```

Try varying the truth assignments, to see what happens. You can add your own formulas to the end of the input, and evaluate them as well. Try to find truth assignments that make each of the formulas tested above evaluate to true. For an extra challenge, try finding a single truth assignment that makes them all true at the same time.

6.2 Truth Tables

The second and third semantic questions we asked are a little trickier than the first. Given a formula A, is there any truth assignment that makes A true? If so, which truth assignments make A true? Instead of considering one particular truth assignment, these questions ask us to quantify over *all* possible truth assignments.

Of course, the number of possible truth assignments depends on the number of propositional letters we're considering. Since each letter has two possible values, n letters will produce 2^n possible truth assignments. This number grows very quickly, so we'll mostly look at smaller formulas here.

We'll use something called a *truth table* to figure out when, if ever, a formula is true. On the left hand side of the truth table, we'll put all of the possible truth assignments for the present propositional letters. On the right hand side, we'll put the truth value of the entire formula under the corresponding assignment.

To begin with, truth tables can be used to concisely summarize the semantics of our logical connectives:

A	B	$A \wedge B$
T	**T**	**T**
T	**F**	**F**
F	**T**	**F**
F	**F**	**F**

A	B	$A \vee B$
T	**T**	**T**
T	**F**	**T**
F	**T**	**T**
F	**F**	**F**

A	B	$A \to B$
T	**T**	**T**
T	**F**	**F**
F	**T**	**T**
F	**F**	**T**

We will leave it to you to write the table for $\neg A$, as an easy exercise.

For compound formulas, the style is much the same. Sometimes it can be helpful to include intermediate columns with the truth values of subformulas:

A	B	C	$A \to B$	$B \to C$	$(A \to B) \vee (B \to C)$
T	**T**	**T**	**T**	**T**	**T**
T	**T**	**F**	**T**	**F**	**T**
T	**F**	**T**	**F**	**T**	**T**
T	**F**	**F**	**F**	**T**	**T**
F	**T**	**T**	**T**	**T**	**T**
F	**T**	**F**	**T**	**F**	**T**
F	**F**	**T**	**T**	**T**	**T**
F	**F**	**F**	**T**	**T**	**T**

By writing out the truth table for a formula, we can glance at the rows and see which truth assignments make the formula true. If all the entries in the final column are **T**, as in the above example, the formula is said to be *valid*.

6.3 Soundness and Completeness

Fix a deductive system, such as natural deduction. A propositional formula is said to be *provable* if there is a formal proof of it in the system. A propositional formula is said to be a *tautology*, or *valid*, if it is true under any truth assignment. Provability is a syntactic notion, insofar as it asserts the existence of a syntactic object, namely, a proof. Validity is a semantic notion, insofar as it has to do with truth assignments and valuations. But, intuitively, these notions should coincide: both express the idea that a formula A *has* to be true, or is *necessarily* true, and one would expect a good proof system to enable us to derive the valid formulas.

Because of the way we have chosen our inference rules and defined the notion of a valuation, this intuition holds true. The statement that every provable formula is valid is known as *soundness*, and the statement that we can prove every valid formula is known as *completeness*.

These notions extend to provability from hypotheses. If Γ is a set of propositional formulas and A is a propositional formula, then A is said to be a *logical consequence* of Γ if, given any truth assignment that makes every formula in Γ true, A is true as well. In this extended setting, soundness says that if A is provable from Γ, then A is a logical consequence of Γ. Completeness runs the other way: if A is a logical consequence of Γ, it is provable from Γ.

Notice that with the rules of natural deduction, a formula A is provable from a set of hypotheses $\{B_1, B_2, \ldots, B_n\}$ if and only if the formula $B_1 \wedge B_2 \wedge \cdots \wedge B_n \to A$ is provable outright, that is, from no hypotheses. So, at least for finite sets of formulas Γ, the two statements of soundness and completeness are equivalent.

Proving soundness and completeness belongs to the realm of *metatheory*, since it requires us to reason about our methods of reasoning. This is not a central focus of this book: we are more concerned with *using* logic and the notion of truth than with establishing their properties. But the notions of soundness and completeness play an important role in helping us understand the nature of the logical notions, and so we will try to provide some hints here as to why these properties hold for propositional logic.

Proving soundness is easier than proving completeness. We wish to show that whenever A is provable from a set of hypotheses, Γ, then A is a logical consequence of Γ. In a later chapter, we will consider proofs by induction, which allows us to establish a property holds of a general collection of objects by showing that it holds of some "simple" ones and is preserved under the passage to objects that are more complex. In the case of natural deduction, it is enough to show that soundness holds of the most basic proofs — using the assumption rule — and that it is preserved under each rule of inference. The base case is easy: the assumption rule says that A is provable from hypothesis A, and clearly every truth assignment that makes A true makes A true. The inductive steps are not much harder; it involves checking that the rules we have chosen mesh with the semantic notions. For example, suppose the last rule is the and introduction rule. In that case, we have a proof of A from some hypotheses Γ, and a proof of B from some hypotheses Δ, and we combine these to form a proof of $A \wedge B$ from the hypotheses in $\Gamma \cup \Delta$, that is, the hypotheses in both. Inductively, we can assume that A is a logical consequence of Γ and that B is a logical consequence of Δ. Let v be any truth assignment that makes every formula in $\Gamma \cup \Delta$ true. Then by the inductive hypothesis, we have that it makes A true, and B true as well. By the definition of the valuation function, $\bar{v}(A \wedge B) = \mathbf{T}$, as required.

Proving completeness is harder. It suffices to show that if A is any tautology, then A is provable. One strategy is to show that natural deduction can simulate the method of truth tables. For example, suppose A is build up from propositional variables B and C. Then in natural deduction, we should be able to prove

$$(B \wedge C) \vee (B \wedge \neg C) \vee (\neg B \wedge C) \vee (\neg B \wedge \neg C),$$

with one disjunct for each line of the truth table. Then, we should be able to use each disjunct to "evaluate" each expression occurring in A, proving it true or false in accordance with its valuation, until we have a proof of A itself.

A nicer way to proceed is to express the rules of natural deduction in a way that allows us to work backwards from A in search of a proof. In other words, first, we give a procedure for constructing a derivation of A by working backwards from A. Then we argue that if the procedure fails, then, at the point where it fails, we can find a truth assignment that makes A false. As a result, if every truth assignment makes A true, the procedure returns a proof of A.

6.4 Exercises

1. Show that $A \to B$, $\neg A \vee B$, and $\neg(A \wedge \neg B)$ are logically equivalent, by writing out the truth table and showing that they have the same values for all truth assignments.
2. Write out the truth table for $(A \to B) \wedge (B \wedge C \to A)$.
3. Show that $A \to B$ and $\neg B \to \neg A$ are equivalent, by writing out the truth tables and showing that they have the same values for all truth assignments.
4. Does the following entailment hold?

   $$\{A \to B \vee C, \neg B \to \neg C\} \models A \to B$$

   Justify your answer by writing out the truth table (sorry, it is long). Indicate clearly the rows where both hypotheses come out true.

CHAPTER

SEVEN

FIRST ORDER LOGIC

Propositional logic provides a good start at describing the general principles of logical reasoning, but it does not go far enough. Some of the limitations are apparent even in the "Malice and Alice" example from Chapter 2. Propositional logic does not give us the means to express a general principle that tells us that if Alice is with her son on the beach, then her son is with Alice; the general fact that no child is younger than his or her parent; or the general fact that if someone is alone, they are not with someone else. To express principles like these, we need a way to talk about objects and individuals, as well as their properties and the relationships between them. These are exactly what is provided by a more expressive logical framework known as *first-order logic*, which will be the topic of the next few chapters.

7.1 Functions, Predicates, and Relations

Consider some ordinary statements about the natural numbers:

- Every natural number is even or odd, but not both.
- A natural number is even if and only if it is divisible by two.
- If some natural number, x, is even, then so is x^2.
- A natural number x is even if and only if $x + 1$ is odd.
- Any prime number that is greater than 2 is odd.
- For any three natural numbers x, y, and z, if x divides y and y divides z, then x divides z.

These statements are true, but we generally do not think of them as *logically valid*: they depend on assumptions about the natural numbers, the meaning of the terms "even" and "odd," and so on. But once we accept the first statement, for example, it seems to be a logical consequence that the number of stairs in the White House is either even or odd, and, in particular, if it is not even, it is odd. To make sense of inferences like these, we need a logical system that can deal with objects, their properties, and relations between them.

Rather than fix a single language once and for all, first-order logic allows us to specify the symbols we wish to use for any given domain of interest. In this section, we will use the following running example:

- the domain of interest is the natural numbers, $\mathbb{N}$.
- there are objects, 0, 1, 2, 3, ….
- there are functions, addition and multiplication, as well as the square function, on this domain.
- there are predicates on this domain, "even," "odd," and "prime."
- there are relations between elements of this domain, "equal," "less than", and "divides."

For our logical language, we will choose symbols 1, 2, 3, *add*, *mul*, *square*, *even*, *odd*, *prime*, *lt*, and so on, to denote these things. We will also have variables x, y, and z ranging over the natural numbers. Note all of the following.

- Functions can take any number of arguments: if x and y are natural numbers, it makes sense to write $\mathit{mul}(x, y)$ and $\mathit{square}(x)$. so *mul* takes two arguments, and *square* takes only one.
- Predicates and relations can also be understood in these terms. The predicates $\mathit{even}(x)$ and $\mathit{prime}(x)$ take one argument, while the binary relations $\mathit{divides}(x, y)$ and $\mathit{lt}(x, y)$ take two arguments.
- Functions are different from predicates! A function takes one or more arguments, and returns a *value*. A predicate takes one or more arguments, and is either true or false. We can think of predicates as returning propositions, rather than values.
- In fact, we can think of the constant symbols $1, 2, 3, \ldots$ as special sorts of function symbols that take zero arguments. Analogously, we can consider the predicates that take zero arguments to be the constant logical values, $\top$ and $\bot$.
- In ordinary mathematics, we often use "infix" notation for binary functions and relations. For example, we usually write $x \times y$ or $x \cdot y$ instead of $\mathit{mul}(x, y)$, and we write $x < y$ instead of $\mathit{lt}(x, y)$. We will use these conventions when writing proofs in natural deduction, and they are supported in Lean as well.
- We will treat the equality relation, $x = y$, as a special binary relation that is included in every first-order language.

First-order logic allows us to build complex expressions out of the basic ones. Starting with the variables and constants, we can use the function symbols to build up compound expressions like these:

$$x + y + z, \quad (x + 1) \times y \times y, \quad \mathit{square}(x + y \times z)$$

Such expressions are called "terms." Intuitively, they name objects in the intended domain of discourse.

Now, using the predicates and relation symbols, we can make assertions about these expressions:

$$\mathit{even}(x + y + z), \quad \mathit{prime}((x + 1) \times y \times y), \quad \mathit{square}(x + y \times z) = w, \quad x + y < z$$

Even more interestingly, we can use propositional connectives to build compound expressions like these:

- $\mathit{even}(x + y + z) \wedge \mathit{prime}((x + 1) \times y \times y)$
- $\neg(\mathit{square}(x + y \times z) = w) \vee x + y < z$
- $x < y \wedge \mathit{even}(x) \wedge \mathit{even}(y) \to x + 1 < y$

The second one, for example, asserts that either $(x+yz)^2$ is not equal to w, or $x+y$ is less than z. Remember, these are expressions in symbolic logic; in ordinary mathematics, we would express the notions using words like "is even" and "if and only if," as we did above. We will use notation like this whenever we are in the realm of symbolic logic, for example, when we write proofs in natural deduction. Expressions like these are called *formulas*. In contrast to terms, which name things, formulas *say things*; in other words, they make assertions about objects in the domain of discourse.

7.2 The Universal Quantifier

What makes first-order logic powerful is that it allows us to make general assertions using *quantifiers*. The universal quantifier $\forall$ followed by a variable x is meant to represent the phrase "for every x." In other words, it asserts that every value of x has the property that follows it. Using the universal quantifier, the examples with which we began this previous section can be expressed as follows:

- $\forall x \; ((\mathit{even}(x) \vee \mathit{odd}(x)) \wedge \neg(\mathit{even}(x) \wedge \neg \mathit{odd}(x)))$

- $\forall x\ (\mathit{even}(x) \leftrightarrow 2 \mid x)$
- $\forall x\ (\mathit{even}(x) \to \mathit{even}(x^2))$
- $\forall x\ (\mathit{even}(x) \leftrightarrow \mathit{odd}(x+1))$
- $\forall x\ (\mathit{prime}(x) \wedge x > 2 \to \mathit{odd}(x))$
- $\forall x\ \forall y\ \forall z (x \mid y \wedge y \mid z \to x \mid z)$

It is common to combine multiple quantifiers of the same kind, and write, for example, $\forall x, y, z\ (x \mid y \wedge y \mid z \to x \mid z)$ in the last expression.

Here are some notes on syntax:

- In symbolic logic, the universal quantifier is usually taken to bind tightly. For example, $\forall x\ P \vee Q$ is interpreted as $(\forall x\ P) \vee Q$, and we would write $\forall x\ (P \vee Q)$ to extend the scope.
- Be careful, however. In other contexts, especially in computer science, people often give quantifiers the *widest* scope possible. This is the case with Lean. For example, `∀ x, P ∨ Q` is interpreted as `∀ x, (P ∨ Q)`, and we would write `(∀ x, P) ∨ Q` to limit the scope.
- After the quantifier $\forall x$, the variable x is *bound*. For example, the expression $\forall x\ (\mathit{even}(x) \vee \mathit{odd}(x))$ is expresses that every number is even or odd. Notice that the variable x does not appear anywhere in the informal statement. The statement is not about x at all; rather x is a dummy variable, a placeholder that stands for the "thing" referred to within a phrase that beings with the words "every thing." We think of the expression $\forall x\ (\mathit{even}(x) \vee \mathit{odd}(x))$ as being the same as the expression $\forall y\ (\mathit{even}(y) \vee \mathit{odd}(y))$. Lean treats these expressions as the same as well.
- In Lean, the expression `∀ x y z, x ∣ y → y ∣ z → x ∣ z` is interpreted as `∀ x y z, x ∣ y → (y ∣ z → x ∣ z)`, with parentheses associated to the *right*. The part of the expression after the universal quantifier can therefore be interpreted as saying "given that `x` divides `y` and that `y` divides `z`, `x` divides `z`." The expression is logically equivalent to `∀ x y z, x ∣ y ∧ y ∣ z → x ∣ z`, but we will see that, in Lean, it is often convenient to express facts like this as an iterated implication.

A variable that is not bound is called *free*. Notice that formulas in first-order logic say things about their free variables. For example, in the interpretation we have in mind, the formula $\forall y\ (x \leq y)$ says that x is less than or equal to every natural number. The formula $\forall z\ (x \leq z)$ says exactly the same thing; we can always rename a bound variable, as long as we pick a name that does not clash with another name that is already in use. On the other hand, the formula $\forall y (w \leq y)$ says that w is less than or equal to every natural number. This is an entirely different statement: it says something about w, rather than x. So renaming a *free* variable changes the meaning of a formula.

Notice also that some formulas, like $\forall x, y\ (x \leq y \vee y \leq x)$, have no free variables at all. Such a formula is called a *sentence*, because it makes an outright assertion, a statement that is either true or false about the intended interpretation. In Chapter 10 we will make the notion of an "intended interpretation" precise, and explain what it means to be "true in an interpretation." For now, the idea that formulas say things about about object in an intended interpretation should motivate the rules for reasoning with such expressions.

In Chapter 1 we proved that the square root of two is irrational. One way to construe the statement is as follows:

> For every pair of natural numbers, a and b, it is not the case that $a^2 = 2b^2$.

The advantage of this formulation is that we can restrict our attention to the natural numbers, without having to consider the larger domain of rationals. In symbolic logic, assuming our intended domain of discourse is the natural numbers, we would express this theorem using the universal quantifier:

$$\forall a, b\ \neg(a^2 = 2b^2).$$

How do we prove such a theorem? Informally, we would use such a pattern:

> Let a and b be arbitrary integers, and suppose $a^2 = 2b^2$.
>
> ...
>
> Contradiction.

What we are really doing is proving that the universal statement holds, by showing that it holds of "arbitrary" values a and b. In natural deduction, the proof would look something like this:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\overline{\;a^2 = 2 \times b^2\;}^{\,1}}{\vdots}}{\bot}}{\neg(a^2 = 2 \times b^2)}^{\,1}}{\forall b \neg(a^2 = 2 \times b^2)}}{\forall a\ \forall b\ \neg(a^2 = 2 \times b^2)}$$

Notice that after the hypothesis is canceled, we have proved $\neg(a^2 = 2 \times b^2)$ without making any assumptions about a and b; at this stage in the proof, they are "arbitrary," justifying the application of the universal quantifiers in the next two rules.

This example motivates the following rule in natural deduction:

$$\dfrac{A(x)}{\forall x\ A(x)}$$

provided x is not free in any uncanceled hypothesis. Here $A(x)$ stands for any formula that (potentially) mentions x. Also remember that if y is any "fresh" variable that does not occur in A, we are thinking of $\forall x\ vA(x)$ as being the same as $\forall y\ A(y)$.

What about the elimination rule? Suppose we know that every number is even or odd. Then, in an ordinary proof, we are free to assert "a is even or a is odd," or "a^2 is even or a^2 is odd." In terms of symbolic logic, this amounts to the following inference: from $\forall x\ (\mathit{even}(x) \vee \mathit{odd}(x))$, we can conclude $\mathit{even}(t) \vee \mathit{odd}(t)$ for any term t. This motivates the elimination rule for the universal quantifier:

$$\dfrac{\forall x A(x)}{A(t)}$$

where t is an arbitrary term.

In a sense, this feels like the elimination rule for implication; we might read the hypothesis as saying "if x is any thing, then x is even or odd." The conclusion is obtained by applying it to the fact that n is a thing. Note that, in general, we could replace n by any *term* in the language, like $n(m + 5) + 2$. Similarly, the introduction rule feels like the introduction rule for implication. If we want to show that everything has a certain property, we temporarily let x denote an arbitrary thing, and then show that it has the relevant property.

7.3 The Existential Quantifier

Dual to the universal quantifier is the existential quantifier, $\exists$, which is used to express assertions such as "some number is even," or, "between any two even numbers there is an odd number."

The following statements about the natural numbers assert the existence of some natural number:

- There exists an odd composite number. (Remember that a natural number is *composite* if it is greater than 1 and not prime.)
- Every natural number greater than one has a prime divisor.

- For every n, if n has a prime divisor smaller than n, then n is composite.

These statements can be expressed in first-order logic using the existential quantifier as follows:

- $\exists n\ (odd(n) \wedge composite(n))$
- $\forall n\ (n > 1 \to \exists p\ (prime(p) \wedge p \mid n))$
- $\forall n\ ((\exists p\ (p \mid n \wedge prime(p) \wedge p < n)) \to composite(n))$

After we write $\exists n$, the variable n is bound in the formula, just as for the universal quantifier. So the formulas $\exists n\ composite(n)$ and $\exists m\ composite(m)$ are considered the same.

How do we prove such existential statements? Suppose we want to prove that there exists an odd composite number. To do this, we just present a candidate, and show that the candidate satisfies the required properties. For example, we could choose 15, and then show that 15 is odd and that 15 is prime. Of course, there's nothing special about 15, and we could have proven it also using a different number, like 9 or 35. The choice of candidate does not matter, as long as it has the required property.

In a natural deduction proof this would look like this:

$$\frac{\begin{array}{c}\vdots\\ odd(15) \wedge composite(15)\end{array}}{\exists n(odd(n) \wedge composite(n))}$$

This illustrates the introduction rule for the existential quantifier:

$$\frac{A(t)}{\exists x A(x)}$$

where t is any term. So to prove an existential formula, we just have to give one particular term for which we can prove that formula. Such term is called a *witness* for the formula.

What about the elimination rule? Suppose that we know that n is some natural number and we know that there exists a prime p such that $p < n$ and $p \mid n$. How can we use this to prove that n is composite? We can reason as follows:

> Let p be any prime such that $p < n$ and $p \mid n$.
>
> ...
>
> Therefore, n is composite.

First, we assume that there is some p which satisfies the properties p is prime, $p < n$ and $p \mid n$, and then we reason about that p. As with case-based reasoning using "or," the assumption is only temporary: if we can show that n is composite from that assumption, that we have essentially shown that n is composite assuming the existence of such a p. Notice that in this pattern of reasoning, p should be "arbitrary." In other words, we should not have assumed anything about p beforehand, we should not make any additional assumptions about p along the way, and the conclusion should not mention p. Only then does it makes sense to say that the conclusion follows from the "mere" existence of a p with the assumed properties.

In natural deduction, the elimination rule is expressed as follows:

$$\frac{\exists x A(x) \qquad \begin{array}{c}\overline{A(y)}^{\,1}\\ \vdots\\ B\end{array}}{B}\,1$$

Here we require that y is not free in B, and that the only uncanceled hypotheses where y occurs freely are the hypotheses $A(y)$ that are canceled when you apply this rule. Formally, this is what it means to say that

y is "arbitrary." As was the case for or elimination and implication introduction, you can use the hypothesis $A(y)$ multiple times in the proof of B, and cancel all of them at once.

Intuitively, the rule says that you can prove B from the assumption $\exists x A(x)$ by assuming $A(y)$ for a fresh variable y, and concluding, in any number of steps, that B follows. You should compare this rule to the rule for or elimination, which is somewhat analogous. In the following example, we show that if $A(x)$ always implies $\neg B(x)$, then there cannot be an x for which both $A(x)$ and $B(x)$ holds.

7.4 Relativization and Sorts

In first-order logic as we have presented it, there is one intended "universe" of objects of discourse, and the universal and existential quantifiers range over that universe. For example, we could design a language to talk about people living in a certain town, with a relation $\mathit{loves}(x, y)$ to express that x loves y. In such a language, we might express the statement that "everyone loves someone" by writing $\forall x \; \exists y \; \mathit{loves}(x, y)$.

You should keep in mind that, at this stage, *loves* is just a symbol. We have designed the language with a certain interpretation in mind, but one could also interpret the language as making statements about the natural numbers, where $\mathit{loves}(x, y)$ means that x is less than or equal to y. In that interpretation, the sentence

$$\forall x, y, z \; (\mathit{loves}(x, y) \wedge \mathit{loves}(y, z) \to \mathit{loves}(x, z))$$

is true, though in the original interpretation it makes an implausible claim about the nature of love triangles. In Chapter 10, we will spell out the notion that the deductive rules of first-order logic enable us to determine the statements that are true in *all* interpretations, just as the rules of propositional logic enable us to determine the statements that are true under all truth assignments.

Returning to the original example, suppose we want to represent the statement that, in our town, all the women are strong and all the men are good looking. We could do that with the following two sentences:

- $\forall x \; (\mathit{woman}(x) \to \mathit{strong}(x))$
- $\forall x \; (\mathit{man}(x) \to \mathit{good\text{-}looking}(x))$

These are instances of *relativization*. The universal quantifier ranges over all the people in the town, but this device gives us a way of using implication to restrict the scope of our statements to men and women, respectively. The trick also comes into play when we render "every prime number greater than two is odd":

$$\forall x \; (\mathit{prime}(x) \wedge x \geq 2 \to \mathit{odd}(x)).$$

We could also read this more literally as saying "for every number x, if x is prime and x is greater than or equal to 2, then x is odd," but it is natural to read it as a restricted quantifier.

It is also possible to relativize the existential quantifier to say things like "some woman is strong" and "some man is good-looking." These are expressed as follows:

- $\exists x \; (\mathit{woman}(x) \wedge \mathit{strong}(x))$
- $\exists x \; (\mathit{man}(x) \wedge \mathit{good\text{-}looking}(x))$

Notice that although we used implication to relativize the universal quantifier, here we need to use conjunction instead of implication. The expression $\exists x \; (\mathit{woman}(x) \to \mathit{strong}(x))$ says that there is something with the property that if it is a woman, then it is strong. Classically this is equivalent to saying that there is something which is either not a woman or is strong, which is a funny thing to say.

Now, suppose we are studying geometry, and we want to express the fact that given any two distinct points p and q and any two lines L and M, if L and M both pass through p and q, then they have to be the same.

(In other words, there is at most one line between two distinct points.) One option is to design a first-order logic where the intended universe is big enough to include both points and lines, and use relativization:

$$\forall p, q, L, M\, (\mathit{point}(p) \wedge \mathit{point}(q) \wedge \mathit{line}(L) \wedge \mathit{line}(M) \\ \wedge \mathit{on}(p, L) \wedge \mathit{on}(q, L) \wedge \mathit{on}(p, M) \wedge \mathit{on}(q, M) \to L = M)$$

But dealing with such predicates is tedious, and there is a mild extension of first-order logic, called *many-sorted first-order logic*, which builds in some of the bookkeeping. In many-sorted logic, one can have different sorts of objects — such as points and lines — and a separate stock of variables and quantifiers ranging over each. Moreover, the specification of function symbols and predicate symbols indicates what sorts of arguments they expect, and, in the case of function symbols, what sort of argument they return. For example, we might choose to have a sort with variables $p, q, r, \ldots$ ranging over points, a sort with variables $L, M, N, \ldots$ ranging over lines, and a relation $\mathit{on}(p, L)$ relating the two. Then the assertion above is rendered more simply as follows:

$$\forall p, q, L, M\ (\mathit{on}(p, L) \wedge \mathit{on}(q, L) \wedge \mathit{on}(p, M) \wedge \mathit{on}(q, M) \to L = M)$$

7.5 Equality

In symbolic logic, we use the expression $s = t$ to express the fact that s and t are "equal" or "identical." The equality symbol is meant to model what we mean when we say, for example, "Alice's brother is the victim," or "2 + 2 = 4." We are asserting that two different descriptions refer to the same object. Because the notion of identity can be applied to virtually any domain of objects, it is viewed as falling under the province of logic.

Talk of "equality" or "identity" raises messy philosophical questions, however. Am I the same person I was three days ago? Are the two copies of *Huckleberry Finn* sitting on my shelf the same book, or two different books? Using symbolic logic to model identity presupposes that we have in mind a certain way of carving up and interpreting the world. We assume that our terms refer to distinct entities, and writing $s = t$ asserts that the two expressions refer to the same thing. Axiomatically, we assume that equality satisfies the following three properties:

- *reflexivity*: $t = t$, for any term t
- *symmetry*: if $s = t$, then $t = s$
- *transitivity*: if $r = s$ and $s = t$, then $r = t$.

These properties are not enough to characterize equality, however. If two expressions denote the same thing, then we should be able to substitute one for any other in any expression. It is convenient to adopt the following convention: if r is any term, we may write $r(x)$ to indicate that the variable x may occur in r. Then, if s is another term, we can thereafter write $r(s)$ to denote the result of replacing s for x in r. The substitution rule for terms thus reads as follows: if $s = t$, then $r(s) = r(t)$.

We already adopted a similar convention for formulas: if we introduce a formula as $A(x)$, then $A(t)$ denotes the result of substituting t for x in A. With this in mind, we can write the rules for equality as follows:

$$\frac{}{t = t} \qquad \frac{s = t}{t = s} \qquad \frac{r = s \qquad s = t}{r = t}$$

$$\frac{s = t}{r(s) = r(t)} \qquad \frac{s = t \qquad P(s)}{P(t)}$$

In the next chapter, you will learn how to use them.

Using equality, we can define even more quantifiers.

- We can express "there are at least two elements x such that $A(x)$ holds" as $\exists x\ \exists y\ (x \neq y \wedge A(x) \wedge A(y))$.

- We can express "there are at most two elements x such that $A(x)$ holds" as $\forall x \; \forall y \; \forall z \; (A(x) \wedge A(y) \wedge A(z) \to x = y \vee y = z \vee x = z)$. This states that if we have three elements a for which $A(a)$ holds, then two of them must be equal.
- We can express "there are exactly two elements x such that $A(x)$ holds" as the conjunction of the above two statements.

As an exercise, write out in first order logic the statements that there are at least, at most, and exactly three elements x such that $A(x)$ holds.

In logic, the expression $\exists! x \; A(x)$ is used to express the fact that there is a *unique* x satisfying $A(x)$, which is to say, there is exactly one such x. As above, this can be expressed as follows:

$$\exists x \; A(x) \wedge \forall y \; \forall y' \; (A(y) \wedge A(y') \to y = y')$$

The first conjunct says that there is at least one object satisfying A, and the second conjunct says that there is at most one. The same thing can be expressed more concisely as follows:

$$\exists x \; (A(x) \wedge \forall y \; (A(y) \to y = x))$$

You should think about why this second expression works. In the next chapter we will see that, using the rules of natural deduction, we can prove that these two expressions are equivalent.

7.6 Exercises

1. A *perfect number* is a number that is equal to the sum of its proper divisors, that is, the numbers that divide it, other than itself. For example, 6 is perfect, because $6 = 1 + 2 + 3$.

   Using a language with variables ranging over the natural numbers and suitable functions and predicates, write down first-order sentences asserting the following. Use a predicate *perfect* to express that a number is perfect.

   (a) 28 is perfect.

   (b) There are no perfect numbers between 100 and 200.

   (c) There are (at least) two perfect numbers between 200 and 10,000. (Express this by saying that there are perfect numbers x and y between 200 and 10,000, with the property that $x \neq y$.)

   (d) Every perfect number is even.

   (e) For every number, there is a perfect number that is larger than it. (This is one way to express the statement that there are infinitely many perfect numbers.)

   Here, the phrase "between a and b" is meant to include a and b.

   By the way, we do not know whether the last two statements are true. They are open questions.

2. Using a language with variables ranging over people, and predicates $\mathit{trusts}(x, y)$, $\mathit{politician}(x)$, $\mathit{knows}(x, y)$, and $\mathit{related\text{-}to}(x, y)$, and $\mathit{rich}(x)$, write down first-order sentences asserting the following:

   (a) Nobody trusts a politician.

   (b) Anyone who trusts a politician is crazy.

   (c) Everyone knows someone who is related to a politician.

   (d) Everyone who is rich is either a politician or knows a politician.

   In each case, some interpretation may be involved. Notice that writing down a logical expression is one way of helping to clarify the meaning.

CHAPTER

EIGHT

NATURAL DEDUCTION FOR FIRST ORDER LOGIC

8.1 Rules of Inference

In the last chapter, we discussed the language of first-order logic, and the rules that govern their use. We summarize them here:

The universal quantifier:

$$\frac{A(x)}{\forall y\ A(y)}\ \forall\mathrm{I} \qquad \frac{\forall x\ A(x)}{A(t)}\ \forall\mathrm{E}$$

In the introduction rule, x should not be free in any uncanceled hypothesis. In the elimination rule, t can be any term that does not clash with any of the bound variables in A.

The existential quantifier:

$$\frac{A(t)}{\exists x A(x)}\ \exists\mathrm{I} \qquad \frac{\exists x A(x) \qquad \begin{matrix}\overline{A(y)}^{\,1} \\ \vdots \\ B\end{matrix}}{B}\ 1\ \exists\mathrm{E}$$

In the introduction rule, t cant be any term that does not clash with any of the bound variables in A. In the elimination rule, y should not be free in B or any uncanceled hypothesis.

Equality:

$$\frac{}{t = t}\ \mathrm{refl} \qquad \frac{s = t}{t = s}\ \mathrm{symm} \qquad \frac{r = s \qquad s = t}{r = t}\ \mathrm{trans}$$

$$\frac{s = t}{r(s) = r(t)}\ \mathrm{subst} \qquad \frac{s = t \qquad P(s)}{P(t)}\ \mathrm{subst}$$

Strictly speaking, only refl and the second substitution rule are necessary. The others can be derived from them.

8.2 The Universal Quantifier

The following example of a proof in natural deduction shows that if, for every x, $A(x)$ holds, and for every x, $B(x)$ holds, then for every x, they both hold:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\forall x\ A(x)}{}^{1}}{A(y)} \qquad \dfrac{\dfrac{}{\forall x\ B(x)}{}^{2}}{B(y)}}{A(y) \wedge B(y)}}{\forall y\ (A(y) \wedge B(y))}}{\forall x\ B(x) \to \forall y\ (A(y) \wedge B(y))}{}^{2}}{\forall x\ A(x) \to (\forall x\ B(x) \to \forall y\ (A(y) \wedge B(y)))}{}^{1}$$

Notice that neither of the assumptions 1 or 2 mention y, so that y is really "arbitrary" at the point where the universal quantifiers are introduced.

Here is another example:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\forall x\ A(x)}{}^{1}}{A(y)}}{A(y) \vee B(y)}}{\forall x\ (A(x) \vee B(x))}}{\forall x\ A(x) \to \forall x\ (A(x) \vee B(x))}{}^{1}$$

As an exercise, try proving the following:

$$\forall x\ (A(x) \to B(x)) \to (\forall x\ A(x) \to \forall x\ B(x)).$$

Here is a more challenging exercise. Suppose I tell you that, in a town, there is a (male) barber that shaves all and only the men who do not shave themselves. You can show that this is a contradiction, arguing informally, as follows:

> By the assumption, the barber shaves himself if and only if he does not shave himself. Call this statement (*).
>
> Suppose the barber shaves himself. By (*), this implies that he does not shave himself, a contradiction. So, the barber does not shave himself.
>
> But using (*) again, this implies that the barber shaves himself, which contradicts the fact we just showed, namely, that the barber does not shave himself.

Try to turn this into a formal argument in natural deduction.

Let us return to the example of the natural numbers, to see how deductive notions play out there. Suppose we have defined *even* and *odd* in such a way that we can prove:

- $\forall n\ (\neg even(n) \to odd(n))$
- $\forall n\ (odd(n) \to \neg even(n))$

Then we can go on to derive $\forall n\ (even(n) \vee odd(n))$ as follows:

$$\dfrac{\dfrac{\dfrac{}{even(n) \vee \neg even(n)} \qquad \dfrac{\dfrac{}{even(n)}{}^{1}}{even(n) \vee odd(n)} \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{\forall n\ (\neg even(n) \to odd(n))}}{\neg even(n) \to odd(n)} \qquad \dfrac{}{\neg even(n)}{}^{1}}{odd(n)}}{even(n) \vee odd(n)}}{even(n) \vee odd(n)}{}^{1}}{\forall n\ (even(n) \vee odd(n))}$$

We can also prove and $\forall n\ \neg(even(n) \wedge odd(n))$:

$$\begin{prooftree}
\AxiomC{}
\UnaryInfC{$odd(n) \to \neg even(n)$}
\AxiomC{}
\RightLabel{H}
\UnaryInfC{$even(n) \wedge odd(n)$}
\UnaryInfC{$odd(n)$}
\BinaryInfC{$\neg even(n)$}
\AxiomC{}
\RightLabel{H}
\UnaryInfC{$even(n) \wedge odd(n)$}
\UnaryInfC{$even(n)$}
\BinaryInfC{$\bot$}
\RightLabel{H}
\UnaryInfC{$\neg(even(n) \wedge odd(n))$}
\UnaryInfC{$\forall n\ \neg(even(n) \wedge odd(n))$}
\end{prooftree}$$

As we move from modeling basic rules of inference to modeling actual mathematical proofs, we will tend to shift focus from natural deduction to formal proofs in Lean. Natural deduction has its uses: as a model of logical reasoning, it provides us with a convenient means to study metatheoretic properties such as soundness and completeness. For working *within* the system, however, proof languages like Lean's tend to scale better, and produce more readable proofs.

8.3 The Existential Quantifier

Remember that the intuition behind the elimination rule for the existential quantifier is that if we know $\exists x\ A(x)$, we can temporarily reason about an arbitrary element y satisfying $A(y)$ in order to prove a conclusion that doesn't depend on y. Here is an example of how it can be used. The next proof says that if we know there is something satisfying both A and B, then we know, in particular, that there is something satisfying A.

$$\begin{prooftree}
\AxiomC{}
\RightLabel{1}
\UnaryInfC{$\exists x(A(x) \wedge B(x))$}
\AxiomC{}
\RightLabel{2}
\UnaryInfC{$A(y) \wedge B(y)$}
\UnaryInfC{$A(y)$}
\UnaryInfC{$\exists x A(x)$}
\RightLabel{2}
\BinaryInfC{$\exists x A(x)$}
\RightLabel{1}
\UnaryInfC{$\exists x(A(x) \wedge B(x)) \to \exists x A(x)$}
\end{prooftree}$$

The following proof shows that if there is something satsifying either A or B, then either there is something satisfying A, or there is something satisfying B.

$$\begin{prooftree}
\AxiomC{}
\RightLabel{1}
\UnaryInfC{$\exists x\ (A(x) \vee B(x))$}
\AxiomC{}
\RightLabel{2}
\UnaryInfC{$A(y) \vee B(y)$}
\AxiomC{}
\RightLabel{3}
\UnaryInfC{$A(y)$}
\UnaryInfC{$\exists x\ A(x)$}
\UnaryInfC{$\exists x\ A(x) \vee \exists x\ B(x)$}
\AxiomC{}
\RightLabel{3}
\UnaryInfC{$B(y)$}
\UnaryInfC{$\exists x\ B(x)$}
\UnaryInfC{$\exists x\ A(x) \vee \exists x\ B(x)$}
\RightLabel{3}
\TrinaryInfC{$\exists x\ A(x) \vee \exists x\ B(x)$}
\RightLabel{2}
\BinaryInfC{$\exists x\ A(x) \vee \exists x\ B(x)$}
\RightLabel{1}
\UnaryInfC{$\exists x\ (A(x) \vee B(x)) \to \exists x\ A(x) \vee \exists x\ B(x))$}
\end{prooftree}$$

The following example is more involved:

$$\begin{prooftree}
\AxiomC{}
\RightLabel{2}
\UnaryInfC{$\exists x\ (A(x) \wedge B(x))$}
\AxiomC{}
\RightLabel{1}
\UnaryInfC{$\forall x\ (A(x) \to \neg B(x))$}
\UnaryInfC{$A(x) \to \neg B(x)$}
\AxiomC{}
\RightLabel{3}
\UnaryInfC{$A(x) \wedge B(x)$}
\UnaryInfC{$A(x)$}
\BinaryInfC{$\neg B(x)$}
\AxiomC{}
\RightLabel{3}
\UnaryInfC{$A(x) \wedge B(x)$}
\UnaryInfC{$B(x)$}
\BinaryInfC{$\bot$}
\RightLabel{3}
\BinaryInfC{$\bot$}
\RightLabel{2}
\UnaryInfC{$\neg \exists x\ (A(x) \wedge B(x))$}
\RightLabel{1}
\UnaryInfC{$\forall x\ (A(x) \to \neg B(x)) \to \neg \exists x\ (A(x) \wedge B(x))$}
\end{prooftree}$$

In this proof, the existential elimination rule (the line labeled 3) is used to cancel two hypotheses at the same time. Note that when this rule is applied, the hypothesis $\forall x\ (A(x) \to \neg B(x))$ has not yet been canceled.

So we have to make sure that this formula doesn't contain the variable x freely. But this is o.k., since this hypothesis contains x only as a bound variable.

Another example is that if x does not occur in P, then $\exists x\ P$ is equivalent to P:

$$\dfrac{\dfrac{\overline{\exists x\ P}^{\,1} \qquad \overline{P}^{\,2}}{P}\,2 \qquad \dfrac{\overline{P}^{\,1}}{\exists x\ P}}{\exists x\ P \leftrightarrow P}\,1$$

This short but tricky, so let us go through it carefully. On the left, we assume $\exists x\ P$ to conclude P. We assume P, and now we can immediately cancel this assumption by existential elimination, since x does not occur in P, so it doesn't occur freely in any assumption or in the conclusion. On the right we use existential introduction to conclude $\exists x\ P$ from P.

8.4 Equality

Recall the natural deduction rules for equality:

$$\dfrac{}{t = t} \qquad \dfrac{s = t}{t = s} \qquad \dfrac{r = s \qquad s = t}{r = t}$$

$$\dfrac{s = t}{r(s) = r(t)} \qquad \dfrac{s = t \qquad P(s)}{P(t)}$$

Keep in mind that we have implicitly fixed some first-order language, and r, s, and t are any terms in that language. Recall also that we have adopted the practice of using functional notation with terms. For example, if we think of $r(x)$ as the term $(x + y) \times (z + 0)$ in the language of arithmetic, then $r(0)$ is the term $(0 + y) \times (z + 0)$ and $r(u + v)$ is $((u + v) + y) \times (z + 0)$. So one example of the first inference on the second line is this:

$$\dfrac{u + v = 0}{((u + v) + y) \times (z + 0) = (0 + y) \times (z + 0)}$$

The second axiom on that line is similar, except now $P(x)$ stands for any *formula*, as in the following inference:

$$\dfrac{u + v = 0 \qquad x + (u + v) < y}{x + 0 < y}$$

Notice that we have written the reflexivity axiom, $t = t$, as a rule with no premises. If you use it in a proof, it does not count as a hypothesis; it is built into the logic.

In fact, we can think of the first inference on the second line as a special case of the first. Consider, for example, the formula $((u + v) + y) \times (z + 0) = (x + y) \times (z + 0)$. If we plug $u + v$ in for x, we get an instance of reflexivity. If we plug in 0, we get the conclusion of the first example above. The following is therefore a derivation of the first inference, using only reflexivity and the second substitution rule above:

$$\dfrac{u + v = 0 \qquad \overline{((u + v) + y) \times (z + 0) = ((u + v) + y) \times (z + 0)}}{((u + v) + y) \times (z + 0) = (0 + y) \times (z + 0)}$$

Roughly speaking, we are replacing the second instance of $u + v$ in an instance of reflexivity with 0 to get the conclusion we want.

Equality rules let us carry out calculations in symbolic logic. This typically amounts to using the equality rules we have already discussed, together with a list of general identities. For example, the following identities hold for any real numbers x, y, and z:

- commutativity of addition: $x + y = y + x$
- associativity of addition: $(x + y) + z = x + (y + z)$

- additive identity: $x + 0 = 0 + x = x$
- additive inverse: $-x + x = x + -x = 0$
- multiplicative identity: $x \cdot 1 = 1 \cdot x = x$
- commutativity of multiplication: $x \cdot y = y \cdot x$
- associativity of multiplication: $(x \cdot y) \cdot z = x \cdot (y \cdot z)$
- distributivity: $x \cdot (y + z) = x \cdot y + x \cdot z, \quad (x + y) \cdot z = x \cdot z + y \cdot z$

You should imagine that there are implicit universal quantifiers in front of each statement, asserting that the statement holds for *any* values of x, y, and z. Note that x, y, and z can, in particular, be integers or rational numbers as well. Calculations involving real numbers, rational numbers, or integers generally involve identities like this.

The strategy is to use the elimination rule for the universal quantifier to instantiate general identities, use symmetry, if necessary, to orient an equation in the right direction, and then using the substitution rule for equality to change something in a previous result. For example, here is a natural deduction proof of a simple identity, $\forall x, y, z\ ((x+y)+z = (x+z)+y)$, using only commutativity and associativity of addition. We have taken the liberty of using a brief name to denote the relevant identities, and combining multiple instances of the universal quantifier introduction and elimination rules into a single step.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\text{assoc}}}{(x+z)+y = x+(z+y)}}{x+(z+y) = (x+z)+y} \quad \dfrac{\dfrac{\dfrac{}{\text{comm}}}{y+z = z+y} \quad \dfrac{\dfrac{}{\text{assoc}}}{(x+y)+z = x+(y+z)}}{(x+y)+z = x+(z+y)}}{(x+y)+z = (x+z)+y}}{\forall x, y, z\ ((x+y)+z = (x+z)+y)}$$

There is generally nothing interesting to be learned from carrying out such calculations in natural deduction, but you should try one or two examples to get the hang of it, and then take pleasure in knowing that it is possible.

8.5 Counterexamples and Relativized Quantifiers

Consider the statement:

> Every prime number is odd.

In first-order logic, we could formulate this as $\forall p\ (prime(p) \to odd(p))$. This statement is false, because there is a prime number which is even, namely the number 2. This is called a *counterexample* to the statement.

More generally, given a formula $\forall x\ A(x)$, a counterexample is a value t such that $\neg A(t)$ holds. Such a counterexample shows that the original formula is false, because we have the following equivalence: $\neg\forall x\ A(x) \leftrightarrow \exists x\ \neg A(x)$. So if we find a value t such that $\neg A(t)$ holds, then by the existential introduction rule we can conclude that $\exists x\ \neg A(x)$, and then by the above equivalence we have $\neg\forall x\ A(x)$. Here is a proof of the equivalence:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\neg\forall x\ A(x)}{}^{1} \quad \dfrac{\dfrac{\dfrac{\dfrac{}{\neg(\exists x\ \neg A(x))}{}^{4} \quad \dfrac{\dfrac{}{\neg A(x)}{}^{5}}{\exists x\ \neg A(x)}}{\bot}}{A(x)}{}^{5}}{\forall x\ A(x)}}{\bot}}{\exists x\ \neg A(x)}{}^{4} \quad \dfrac{\dfrac{\dfrac{}{\exists x\ \neg A(x)}{}^{1} \quad \dfrac{\dfrac{}{\neg A(y)}{}^{3} \quad \dfrac{\dfrac{}{\forall x\ A(x)}{}^{2}}{A(y)}}{\bot}}{\bot}{}^{3}}{\neg\forall x\ A(x)}{}^{2}}{\neg\forall x\ A(x) \leftrightarrow \exists x\ \neg A(x)}{}^{1}$$

One remark about the proof: at the step marked by 4 we *cannot* use the existential introduction rule, because at that point our only assumption is $\neg\forall x\ A(x)$, and from that assumption we cannot prove $\neg A(t)$ for a particular term t. So we use a proof by contradiction there.

As an exercise, prove the "dual" equivalence yourself: $\neg\exists x\ A(x) \leftrightarrow \forall x\ \neg A(x)$. This can be done without using proof by contradiction.

In Chapter 7 we saw examples of how to use relativization to restrict the scope of a universal quantifier. Suppose we want to say "every prime number is greater than 1". In first order logic this can be written as $\forall n(\mathit{prime}(n) \to n > 1)$. The reason is that the original statement is equivalent to the statement "for every natural number, if it is prime, then it is greater than 1". Similarly, suppose we want to say "there exists a prime number greater than 100." This is equivalent to saying "there exists a natural number which is prime and greater than 100," which can be expressed as $\exists n\ (\mathit{prime}(n) \wedge n > 100)$.

As an exercise you can prove the above results about negations of quantifiers also for relativized quantifiers. Specifically, prove the following statements:

- $\neg\exists x\ (A(x) \wedge B(x)) \leftrightarrow \forall x\ (A(x) \to \neg B(x))$;
- $\neg\forall x\ (A(x) \to B(x)) \leftrightarrow \exists x(A(x) \wedge \neg B(x))$

For reference, here is a list of valid sentences involving quantifiers:

- $\forall x\ A \leftrightarrow A$ if x is not free in A
- $\exists x\ A \leftrightarrow A$ if x is not free in A
- $\forall x\ (A(x) \wedge B(x)) \leftrightarrow \forall x\ A(x) \wedge \forall x\ B(x)$
- $\exists x\ (A(x) \wedge B) \leftrightarrow \exists\ xA(x) \wedge B$ if x is not free in B
- $\exists x\ (A(x) \vee B(x)) \leftrightarrow \exists\ xA(x) \vee \exists\ xB(x)$
- $\forall x\ (A(x) \vee B) \leftrightarrow \forall x\ A(x) \vee B$ if x is not free in B
- $\forall x\ (A(x) \to B) \leftrightarrow (\exists x\ A(x) \to B)$ if x is not free in B
- $\exists x\ (A(x) \to B) \leftrightarrow (\forall x\ A(x) \to B)$ if x is not free in B
- $\forall x\ (A \to B(x)) \leftrightarrow (A \to \forall x\ B(x))$ if x is not free in A
- $\exists x\ (A(x) \to B) \leftrightarrow (A(x) \to \exists\ xB)$ if x is not free in B
- $\exists x\ A(x) \leftrightarrow \neg\forall x\ \neg A(x)$
- $\forall x\ A(x) \leftrightarrow \neg\exists x\ \neg A(x)$
- $\neg\exists x\ A(x) \leftrightarrow \forall x\ \neg A(x)$
- $\neg\forall x\ A(x) \leftrightarrow \exists x\ \neg A(x)$

All of these can be derived in natural deduction. The last two allow us to push negations inwards, so we can continue to put first-order formulas in negation normal form. Other rules allow us to bring quantifiers to the front of any formula, though, in general, there will be multiple ways of doing this. For example, the formula

$$\forall x\ A(x) \to \exists y\ \forall z\ B(y, z)$$

is equivalent to both

$$\exists x, y\ \forall z\ (A(x) \to B(y, z))$$

and

$$\exists y\ \forall z\ \exists x\ (A(x) \to B(y, z)).$$

A formula with all the quantifiers in front is said to be in *prenex* form.

8.6 Exercises

1. Give a natural deduction proof of

$$\forall x\ (A(x) \to B(x)) \to (\forall x\ A(x) \to \forall x\ B(x)).$$

2. Give a natural deduction proof of $\forall x\ B(x)$ from hypotheses $\forall x\ (A(x) \vee B(x))$ and $\forall y\ \neg A(y)$.
3. From hypotheses $\forall x\ (\mathit{even}(x) \vee \mathit{odd}(x))$ and $\forall x\ (\mathit{odd}(x) \to \mathit{even}(s(x)))$ give a natural deduction proof $\forall x\ (\mathit{even}(x) \vee \mathit{even}(s(x)))$. (It might help to think of $s(x)$ as the function defined by $s(x) = x + 1$.)
4. Give a natural deduction proof of $\exists x\ A(x) \vee \exists x\ B(x) \to \exists x\ (A(x) \vee B(x))$.
5. Give a natural deduction proof of $\exists x\ (A(x) \wedge C(x))$ from the assumptions $\exists x\ (A(x) \wedge B(x))$ and $\forall x\ (A(x) \wedge B(x) \to C(x))$.
6. Prove some of the other equivalences in the last section.
7. Consider some of the various ways of expressing "nobody trusts a politician" in first-order logic:
   - $\forall x\ (\mathit{politician}(x) \to \forall y\ (\neg \mathit{trusts}(y, x)))$
   - $\forall x, y\ (\mathit{politician}(x) \to \neg \mathit{trusts}(y, x))$
   - $\neg \exists x, y\ (\mathit{politician}(x) \wedge \mathit{trusts}(y, x))$
   - $\forall x, y\ (\mathit{trusts}(y, x) \to \neg \mathit{politician}(x))$

   They are all logically equivalent. Show this for the second and the fourth, by giving natural deduction proofs of each from the other. (As a shortcut, in the $\forall$ introduction and elimination rules, you can introduce / eliminate both variables in one step.)
8. Formalize the following statements, and give a natural deduction proof in which the first three statements appear as (uncancelled) hypotheses, and the last line is the conclusion:
   - Every young and healthy person likes baseball.
   - Every active person is healthy.
   - Someone is young and active.
   - Therefore, someone likes baseball.

   Use $Y(x)$ for "is young," $H(x)$ for "is healthy," $A(x)$ for "is active," and $B(x)$ for "likes baseball."
9. Give a natural deduction proof of $\forall x, y, z\ (x = z \to (y = z \to x = y))$ using the equality rules in Section 8.4.
10. Give a natural deduction proof of $\forall x, y\ (x = y \to y = x)$ using only these two hypotheses (and none of the new equality rules):
    - $\forall x\ (x = x)$
    - $\forall u, v, w\ (u = w \to (v = w \to u = v))$

    (Hint: Choose instantiations of u, v, and w carefully. You can instantiate all the universal quantifiers in one step, as on the last homework assignment.)
11. Give a natural deduction proof of $\neg \exists x\ (A(x) \wedge B(x)) \leftrightarrow \forall x\ (A(x) \to \neg B(x))$
12. Give a natural deduction proof of $\neg \forall x\ (A(x) \to B(x)) \leftrightarrow \exists x\ (A(x) \wedge \neg B(x))$
13. Remember that both the following express $\exists! x\ A(x)$, that is, the statement that there is a unique x satisfying $A(x)$:
    - $\exists x\ (A(x) \wedge \forall y\ (A(y) \to y = x))$

- $\exists x\ A(x) \wedge \forall y\ \forall y'\ (A(y) \wedge A(y') \to y = y')$

Do the following:

- Give a natural deduction proof of the second, assuming the first as a hypothesis.
- Give a natural deduction proof of the first, asssuming the second as a hypothesis.

(Warning: these are long.)

CHAPTER

NINE

FIRST ORDER LOGIC IN LEAN

9.1 Functions, Predicates, and Relations

In the last chapter, we discussed the language of first-order logic. We will see in the course of this book that Lean's built-in logic is much more expressive; but it *includes* first-order logic, which is to say, anything that can be expressed (and proved) in first-order logic can be expressed (and proved) in Lean.

Lean is based on a foundational framework called *type theory*, in which every variable is assumed to range elements of some *type*. You can think of a type as being a "universe," or a "domain of discourse," in the sense of first-order logic.

For example, suppose we want to work with a first-order language with one constant symbol, one unary function symbol, one binary function symbol, one unary relation symbol, and one binary relation symbol. We can declare a new type `U` (for "universe") and the relevant symbols as follows:

```
constant U : Type

constant c : U
constant f : U → U
constant g : U → U → U
constant P : U → Prop
constant R : U → U → Prop
```

We can then use them as follows:

```
variables x y : U

#check c
#check f c
#check g x y
#check g x (f c)

#check P (g x (f c))
#check R x y
```

The `#check` command tells us that the first four expressions have type `U`, and that the last two have type `Prop`. Roughly, this means that the first four expressions correspond to terms of first-order logic, and that the last two correspond to formulas.

Note all the following:

- A unary function is represented as an object of type `U → U` and a binary function is represented as an object of type `U → U → U`, using the same notation as for implication between propositions.

- We write, for example, `f x` to denote the result of applying `f` to `x`, and `g x y` to denote the result of applying `g` to `x` and `y`, again just as we did when using modus ponens for first-order logic. Parentheses are needed in the expression `g x (f c)` to ensure that `f c` is parsed as a single argument.
- A unary predicate is presented as an object of type `U → Prop` and a binary function is represented as an object of type `U → U → Prop`. You can think of a binary relation `R` as being a function that assumes two arguments in the universe, `U`, and returns a proposition.
- We write `P x` to denote the assertion that `P` holds of `x`, and `R x y` to denote that `R` holds of `x` and `y`.

You may reasonably wonder what difference there is between a constant and a variable in Lean. The following declarations also work:

```
variable U : Type

variable c : U
variable f : U → U
variable g : U → U → U
variable P : U → Prop
variable R : U → U → Prop

variables x y : U

#check c
#check f c
#check g x y
#check g x (f c)

#check P (g x (f c))
#check R x y
```

Although the examples function in much the same way, the `constant` and `variable` commands do very different things. The `constant` command declares a new object, axiomatically, and adds it to the list of objects Lean knows about. In contrast, when it is first executed, the `variable` command does not create anything. Rather, it tells Lean that whenever we enter an expression using the corresponding identifier, it should create a temporary variable of the corresponding type.

Many types are already declared in Lean's standard library. For example, there is a type written `nat` or ℕ, that denotes the natural numbers:

```
#check nat
#check ℕ
```

You can enter the unicode ℕ with `\nat` or `\N`. The two expressions mean the same thing.

Using this built-in type, we can model the language of arithmetic, as described in the last chapter, as follows:

```
namespace hide

constant mul : ℕ → ℕ → ℕ
constant add : ℕ → ℕ → ℕ
constant square : ℕ → ℕ
constant even : ℕ → Prop
constant odd : ℕ → Prop
constant prime : ℕ → Prop
constant divides : ℕ → ℕ → Prop
constant lt : ℕ → ℕ → Prop
constant zero : ℕ
constant one : ℕ
```

```
end hide
```

We have used the `namespace` command to avoid conflicts with identifiers that are already declared in the Lean library. We can again use the `#check` command to try them out:

```
variables w x y z : ℕ

#check mul x y
#check add x y
#check square x
#check even x
```

In fact, all of the functions, predicates, and relations discussed here, except for the "square" function and "prime," are defined in the core Lean library. They become available to us when we put the commands `import data.nat` and `open nat` at the top of a file in Lean.

```
constant square : ℕ → ℕ
constant prime : ℕ → Prop
constant even : ℕ → Prop

variables w x y z : ℕ

#check even (x + y + z) ∧ prime ((x + 1) * y * y)
#check ¬ (square (x + y * z) = w) ∨ x + y < z
#check x < y ∧ even x ∧ even y → x + 1 < y
```

Here, we declare the constants `square` and `prime` axiomatically, but refer to the other operations and predicates in the Lean library. In this book, we will often proceed in this way, telling you explicitly what facts from the library you should use for exercises.

Again, note the following aspects of syntax:

- In contrast to ordinary mathematical notation, in Lean, functions are applied without parentheses or commas. For example, we write `square x` and `add x y` instead of $square(x)$ and $add(x, y)$.
- The same holds for predicates and relations: we write `even x` and `lt x y` instead of $even(x)$ and $lt(x, y)$, as one might do in symbolic logic.
- The notation `add : ℕ → ℕ → ℕ` indicates that addition assumes two arguments, both natural numbers, and returns a natural number.
- Similarly, the notation `divides : ℕ → ℕ → Prop` indicates that `divides` is a binary relation, which assumes two natural numbers as arguments and forms a proposition. In other words, `divides x y` expresses the assertion that `x` divides `y`.

Lean can help us distinguish between terms and formulas. If we `#check` the expression `x + y + 1` in Lean, we are told it has type ℕ, which is to say, it denotes a natural number. If we `#check` the expression `even (x + y + 1)`, we are told that it has type `Prop`, which is to say, it expresses a proposition.

In Chapter 7 we considered many-sorted logic, where one can have multiple universes. For example, we might want to use first-order logic for geometry, with quantifiers ranging over points and lines. In Lean, we can model this as by introducing a new type for each sort:

```
variables Point Line : Type
variable  lies_on : Point → Line → Prop
```

We can then express that two distinct points determine a line as follows:

```
#check ∀ (p q : Point) (L M : Line),
       p ≠ q → lies_on p L → lies_on q L → lies_on p M → lies_on q M → L = M
```

Notice that we have followed the convention of using iterated implication rather than conjunction in the antecedent. In fact, Lean is smart enough to infer what sorts of objects `p`, `q`, `L`, and `M` are from the fact that they are used with the relation `on`, so we could have written, more simply, this:

```
#check ∀ p q L M, p ≠ q → lies_on p L → lies_on q L → lies_on p M → lies_on q M → L = M
```

9.2 Using the Universal Quantifier

In Lean, you can enter the universal quantifier by writing `\all`. The motivating examples from Section 7.1 are rendered as follows:

```
constant prime : ℕ → Prop
constant even : ℕ → Prop
constant odd : ℕ → Prop

#check ∀ x, (even x ∨ odd x) ∧ ¬ (even x ∧ odd x)
#check ∀ x, even x ↔ 2 ∣ x
#check ∀ x, even x → even (x^2)
#check ∀ x, even x ↔ odd (x + 1)
#check ∀ x, prime x ∧ x > 2 → odd x
#check ∀ x y z, x ∣ y → y ∣ z → x ∣ z
```

Remember that Lean expects a comma after the universal quantifier, and gives it the *widest* scope possible. For example, `∀ x, P ∨ Q` is interpreted as `∀ x, (P ∨ Q)`, and we would write `(∀ x, P) ∨ Q` to limit the scope. If you prefer, you can use the plain ascii expression `forall` instead of the unicode ∀.

In Lean, then, the pattern for proving a universal statement is rendered as follows:

```
variable U : Type
variable P : U → Prop

example : ∀ x, P x :=
assume x,
show P x, from sorry
```

Read `assume x` as "fix and arbitrary value `x` of `U`." Since we are allowed to rename bound variables at will, we can equivalently write either of the following:

```
variable U : Type
variable P : U → Prop

example : ∀ y, P y :=
assume x,
show P x, from sorry

example : ∀ x, P x :=
assume y,
show P y, from sorry
```

This constitutes the introduction rule for the universal quantifier. It is very similar to the introduction rule for implication: instead of using `assume` to temporarily introduce an assumption, we use `assume` to temporarily introduce a new object, `y`. (In fact, `assume` and `assume` are both alternate syntax for a single internal construct in Lean, which can also be denoted by λ.)

The elimination rule is, similarly, implemented as follows:

```
variable U : Type
variable P : U → Prop
variable h : ∀ x, P x
variable a : U

example : P a :=
show P a, from h a
```

Observe the notation: `P a` is obtained by "applying" the hypothesis `h` to `a`. Once again, note the similarity to the elimination rule for implication.

Here is an example of how it is used:

```
variable U : Type
variables A B : U → Prop

example (h1 : ∀ x, A x → B x) (h2 : ∀ x, A x) : ∀ x, B x :=
assume y,
have h3 : A y, from h2 y,
have h4 : A y → B y, from h1 y,
show B y, from h4 h3
```

Here is an even shorter version of the same proof, where we avoid using `have`:

```
example (h1 : ∀ x, A x → B x) (h2 : ∀ x, A x) : ∀ x, B x :=
assume y,
show B y, from h1 y (h2 y)
```

You should talk through the steps, here. Applying `h1` to `y` yields a proof of `A y → B y`, which we then apply to `h2 y`, which is a proof of `A y`. The result is the proof of `B y` that we are after.

In the last chapter, we considered the following proof in natural deduction:

$$
\begin{prooftree}
\AxiomC{}
\RightLabel{^{1}}
\UnaryInfC{$\forall x\, A(x)$}
\UnaryInfC{$A(y)$}
\AxiomC{}
\RightLabel{^{2}}
\UnaryInfC{$\forall x\, B(x)$}
\UnaryInfC{$B(y)$}
\BinaryInfC{$A(y) \wedge B(y)$}
\UnaryInfC{$\forall y\, (A(y) \wedge B(y))$}
\RightLabel{^{2}}
\UnaryInfC{$\forall x\, B(x) \to \forall y\, (A(y) \wedge B(y))$}
\RightLabel{^{1}}
\UnaryInfC{$\forall x\, A(x) \to (\forall x\, B(x) \to \forall y\, (A(y) \wedge B(y)))$}
\end{prooftree}
$$

Here is the same proof rendered in Lean:

```
variable U : Type
variables A B : U → Prop

example : (∀ x, A x) → (∀ x, B x) → (∀ x, A x ∧ B x) :=
assume hA : ∀ x, A x,
assume hB : ∀ x, B x,
assume y,
have Ay : A y, from hA y,
have By : B y, from hB y,
show A y ∧ B y, from and.intro Ay By
```

Here is an alternative version, using the "anonymous" versions of `have`:

```
variable U : Type
variables A B : U → Prop

example : (∀ x, A x) → (∀ x, B x) → (∀ x, A x ∧ B x) :=
assume hA : ∀ x, A x,
assume hB : ∀ x, B x,
assume y,
have A y, from hA y,
have B y, from hB y,
show A y ∧ B y, from and.intro ‹A y› ‹B y›
```

The exercises below ask you to prove the barber paradox, which was discussed in the last chapter. You can do that using only propositional reasoning and the rules for the universal quantifer that we have just discussed.

9.3 Using the Existential Quantifier

In Lean, you can type the existential quantifier, ∃, by writing `\ex`. If you prefer you can use the ascii equivalent, `exists`. The introduction rule is `exists.intro` and requires two arguments: a term, and a proof that that term satisfies the required property.

```
variable U : Type
variable P : U → Prop

example (y : U) (h : P y) : ∃ x, P x :=
exists.intro y h
```

The elimination rule for the existential quantifier is given by `exists.elim`. It follows the form of the natural deduction rule: if we know `∃x, P x` and we are trying to prove `Q`, it suffices to introduce a new variable, `y`, and prove `Q` under the assumption that `P y` holds.

```
variable U : Type
variable P : U → Prop
variable Q : Prop

example (h1 : ∃ x, P x) (h2 : ∀ x, P x → Q) : Q :=
exists.elim h1
  (assume (y : U) (h : P y),
    have h3 : P y → Q, from h2 y,
    show Q, from h3 h)
```

The following example uses both the introduction and the elimination rules for the existential quantifier.

```
variable U : Type
variables A B : U → Prop

example : (∃ x, A x ∧ B x) → ∃ x, A x :=
assume h1 : ∃ x, A x ∧ B x,
exists.elim h1
  (assume y (h2 : A y ∧ B y),
    have h3 : A y, from and.left h2,
    show ∃ x, A x, from exists.intro y h3)
```

Notice the parentheses in the hypothesis; if we left them out, everything after the first `∃ x` would be included in the scope of that quantifier. From the hypothesis, we obtain a `y` that satisfies `A y ∧ B y`, and hence `A y` in particular. So `y` is enough to witness the conclusion.

It is sometimes annoying to enclose the proof after an `exists.elim` in parenthesis, as we did here with the `assume ... show` block. To avoid that, we can use a bit of syntax from the programming world, and use a dollar sign instead. In Lean, an expression `f $ t` means the same thing as `f (t)`, with the advantage that we do not have to remember to close the parenthesis. With this gadget, we can write the proof above as follows:

```
variable U : Type
variables A B : U → Prop

example : (∃ x, A x ∧ B x) → ∃ x, A x :=
assume h1 : ∃ x, A x ∧ B x,
exists.elim h1 $
assume y (h2 : A y ∧ B y),
have h3 : A y, from and.left h2,
show ∃ x, A x, from exists.intro y h3
```

The following example is more involved:

```
example : (∃ x, A x ∨ B x) → (∃ x, A x) ∨ (∃ x, B x) :=
assume h1 : ∃ x, A x ∨ B x,
exists.elim h1 $
assume y (h2 : A y ∨ B y),
or.elim h2
  (assume h3 : A y,
    have h4 : ∃ x, A x, from exists.intro y h3,
    show (∃ x, A x) ∨ (∃ x, B x), from or.inl h4)
  (assume h3 : B y,
    have h4 : ∃ x, B x, from exists.intro y h3,
    show (∃ x, A x) ∨ (∃ x, B x), from or.inr h4)
```

Note again the placement of parentheses in the statement.

In the last chapter, we considered the following natural deduction proof:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\exists x\,(A(x)\wedge B(x))}\,{}^{2}\qquad \dfrac{\dfrac{\dfrac{\dfrac{}{\forall x\,(A(x)\to\neg B(x))}\,{}^{1}}{A(x)\to\neg B(x)}\qquad\dfrac{\dfrac{}{A(x)\wedge B(x)}\,{}^{3}}{A(x)}}{\neg B(x)}\qquad\dfrac{\dfrac{}{A(x)\wedge B(x)}\,{}^{3}}{B(x)}}{\bot}}{\bot}\,{}^{3}}{\neg\exists x\,(A(x)\wedge B(x))}\,{}^{2}}{\forall x\,(A(x)\to\neg B(x))\to\neg\exists x\,(A(x)\wedge B(x))}\,{}^{1}$$

Here is a proof of the same implication in Lean:

```
variable U : Type
variables A B : U → Prop

example : (∀ x, A x → ¬ B x) → ¬ ∃ x, A x ∧ B x :=
assume h1 : ∀ x, A x → ¬ B x,
assume h2 : ∃ x, A x ∧ B x,
exists.elim h2 $
assume x (h3 : A x ∧ B x),
have h4 : A x, from and.left h3,
have h5 : B x, from and.right h3,
have h6 : ¬ B x, from h1 x h4,
show false, from h6 h5
```

Here, we use `exists.elim` to introduce a value `x` satisfying `A x ∧ B x`. The name is arbitrary; we could just as well have used `z`:

```
example : (∀ x, A x → ¬ B x) → ¬ ∃ x, A x ∧ B x :=
assume h1 : ∀ x, A x → ¬ B x,
assume h2 : ∃ x, A x ∧ B x,
exists.elim h2 $
assume z (h3 : A z ∧ B z),
have h4 : A z, from and.left h3,
have h5 : B z, from and.right h3,
have h6 : ¬ B z, from h1 z h4,
show false, from h6 h5
```

Here is another example of the exists-elimination rule:

```
variable U : Type
variable u : U
variable P : Prop

example : (∃x : U, P) ↔ P :=
iff.intro
  (assume h1 : ∃x, P,
    exists.elim h1 $
    assume x (h2 : P),
    h2)
  (assume h1 : P,
    exists.intro u h1)
```

It is subtle: the proof does not go through if we do not declare a variable `u` of type `U`, even though `u` does not appear in the statement of the theorem. The semantics of first-order logic, discussed in the next chapter, presuppose that the universe is nonempty. In Lean, however, it is possible for a type to be empty, and so the proof above depends on the fact that there is an element `u` in `U`.

These features are all illustrated in the following example:

```
variable U : Type
variables P R : U → Prop
variable Q : Prop

example (h1 : ∃x, P x ∧ R x) (h2 : ∀x, P x → R x → Q) : Q :=
let ⟨y, hPy, hRy⟩ := h1 in
show Q, from h2 y hPy hRy
```

9.4 Equality and calculational proofs

In Lean, reflexivity, symmetry, and transitivity are called `eq.refl`, `eq.symm`, and `eq.trans`, and the second substitution rule is called `eq.subst`. Their uses are illustrated below.

```
variable A : Type

variables x y z : A
```

```
variable P : A → Prop

example : x = x :=
show x = x, from eq.refl x

example : y = x :=
have h : x = y, from sorry,
show y = x, from eq.symm h

example : x = z :=
have h1 : x = y, from sorry,
have h2 : y = z, from sorry,
show x = z, from eq.trans h1 h2

example : P y :=
have h1 : x = y, from sorry,
have h2 : P x, from sorry,
show P y, from eq.subst h1 h2
```

The rule `eq.refl` above assumes `x` as an argument, because there is no hypothesis to infer it from. All the other rules assume their premises as arguments. Here is an example of equational reasoning:

```
variables (A : Type) (x y z : A)

example : y = x → y = z → x = z :=
assume h1 : y = x,
assume h2 : y = z,
have h3 : x = y, from eq.symm h1,
show x = z, from eq.trans h3 h2
```

This proof can be written more concisely:

```
example : y = x → y = z → x = z :=
assume h1 h2, eq.trans (eq.symm h1) h2
```

Because calculation is so important in mathematics, however, Lean provides more efficient ways of carrying them out. One is the `rewrite` tactic. Typing `begin` and `end` in a Lean proof puts Lean into "tactic mode," which means that Lean then expects a list of instructions. The command `rewrite` then uses identities to change the goal. For example, the previous proof could be written as follows:

```
example : y = x → y = z → x = z :=
assume h1 : y = x,
assume h2 : y = z,
show x = z,
  begin
    rewrite ←h1,
    apply h2
  end
```

The `rewrite` tactic can be abbreviated `rw`.

The first command changes the goal `x = z` to `y = z`; the minus sign before `h1` tells Lean to use the equation in the reverse direction. After that, we can finish the goal by applying `h2`.

An alternative is to rewrite the goal using `h1` and `h2`, which reduces the goal to `x = x`. When that happens, `rewrite` automatically applies reflexivity.

```
example : y = x → y = z → x = z :=
assume h1 : y = x,
assume h2 : y = z,
show x = z,
  begin
    rw ←h1,
    rw h2
  end
```

In fact, a sequence of rewrites can be combined, using square brackets:

```
example : y = x → y = z → x = z :=
assume h1 : y = x,
assume h2 : y = z,
show x = z,
  begin
    rw [←h1, h2]
  end
```

And when you reduce a proof to a single tactic, you can use `by` instead of `begin ... end`.

```
example : y = x → y = z → x = z :=
assume h1 : y = x,
assume h2 : y = z,
show x = z, by rw [←h1, h2]
```

We will see in the coming chapters that in ordinary mathematical proofs, one commonly carries out calculations in a format like this:

$$\begin{aligned} t_1 &= t_2 \\ \ldots &= t_3 \\ \ldots &= t_4 \\ \ldots &= t_5 \end{aligned}$$

Lean has a mechanism to model calculational proofs like this. Whenever a proof of an equation is expected, you can provide a proof using the identifier `calc`, following by a chain of equalities and justification, in the following form:

```
calc
  e1 = e2    : justification 1
    ... = e3 : justification 2
    ... = e4 : justification 3
    ... = e5 : justification 4
```

The chain can go on as long as needed. Each justification is the name of the assumption or theorem that is used. For example, the previous proof could be written as follows:

```
example : y = x → y = z → x = z :=
assume h1 : y = x,
assume h2 : y = z,
calc
    x = y : eq.symm h1
  ... = z : h2
```

As usual, the syntax is finicky; notice that there are no commas in the `calc` expression, and the colons and dots need to be entered exactly in that form. All that varies are the expressions `e1, e2, e3, ...` and the justifications themselves.

The `calc` environment is most powerful when used in conjunction with `rewrite`, since we can then rewrite expressions with facts from the library. For example, Lean's library has a number of basic identities for the integers, such as these:

```
variables x y z : int

example : x + 0 = x :=
add_zero x

example : 0 + x = x :=
zero_add x

example : (x + y) + z = x + (y + z) :=
add_assoc x y z

example : x + y = y + x :=
add_comm x y

example : (x * y) * z = x * (y * z) :=
mul_assoc x y z

example : x * y = y * x :=
mul_comm x y

example : x * (y + z) = x * y + x * z :=
left_distrib x y z

example : (x + y) * z = x * z + y * z :=
right_distrib x y z
```

You can also write the type of integers as ℤ, entered with either `\Z` or `\int`. Notice that, for example, `add_comm` is the theorem `∀ x y, x + y = y + x`. So to instantiate it to `s + t = t + s`, you write `add_comm s t`. Using these axioms, here is the calculation above rendered in Lean, as a theorem about the integers:

```
example (x y z : int) : (x + y) + z = (x + z) + y :=
calc
   (x + y) + z = x + (y + z) : add_assoc x y z
           ... = x + (z + y) : eq.subst (add_comm y z) rfl
           ... = (x + z) + y : eq.symm (add_assoc x z y)
```

Using `rewrite` is more efficient, though at times we have to provide information to specify where the rules are used:

```
example (x y z : int) : (x + y) + z = (x + z) + y :=
calc
  (x + y) + z = x + (y + z) : by rw add_assoc
          ... = x + (z + y) : by rw [add_comm y z]
          ... = (x + z) + y : by rw add_assoc
```

In that case, we can use a single `rewrite`:

```
example (x y z : int) : (x + y) + z = (x + z) + y :=
by rw [add_assoc, add_comm y z, add_assoc]
```

If you #check the proof before the sequence of `rewrites` is sufficient, the error message will display the remaining goal.

Here is another example:

```
variables a b d c : int

example : (a + b) * (c + d) = a * c + b * c + a * d + b * d :=
calc
  (a + b) * (c + d) = (a + b) * c + (a + b) * d : by rw left_distrib
    ... = (a * c + b * c) + (a + b) * d         : by rw right_distrib
    ... = (a * c + b * c) + (a * d + b * d)     : by rw right_distrib
    ... = a * c + b * c + a * d + b * d         : by rw ←add_assoc
```

Once again, we can get by with a shorter proof:

```
example : (a + b) * (c + d) = a * c + b * c + a * d + b * d :=
by rw [left_distrib, right_distrib, right_distrib, ←add_assoc]
```

9.5 Exercises

1. Fill in the `sorry`.

```
section
  variable A : Type
  variable f : A → A
  variable P : A → Prop
  variable  h : ∀ x, P x → P (f x)

  -- Show the following:
  example : ∀ y, P y → P (f (f y)) :=
  sorry
end
```

2. Fill in the `sorry`.

```
section
  variable U : Type
  variables A B : U → Prop

  example : (∀ x, A x ∧ B x) → ∀ x, A x :=
  sorry
end
```

3. Fill in the `sorry`.

```
section
  variable U : Type
  variables A B C : U → Prop

  variable h1 : ∀ x, A x ∨ B x
  variable h2 : ∀ x, A x → C x
  variable h3 : ∀ x, B x → C x

  example : ∀ x, C x :=
  sorry
end
```

4. Fill in the `sorry`'s below, to prove the barber paradox.

```
open classical   -- not needed, but you can use it

-- This is an exercise from Chapter 4. Use it as an axiom here.
axiom not_iff_not_self (P : Prop) : ¬ (P ↔ ¬ P)

example (Q : Prop) : ¬ (Q ↔ ¬ Q) :=
not_iff_not_self Q

section
  variable Person : Type
  variable shaves : Person → Person → Prop
  variable barber : Person
  variable h : ∀ x, shaves barber x ↔ ¬ shaves x x

  -- Show the following:
  example : false :=
  sorry
end
```

5. Fill in the sorry.

```
section
  variable U : Type
  variables A B : U → Prop

  example : (∃ x, A x) → ∃ x, A x ∨ B x :=
  sorry
end
```

6. Fill in the sorry.

```
section
  variable U : Type
  variables A B : U → Prop

  variable h1 : ∀ x, A x → B x
  variable h2 : ∃ x, A x

  example : ∃ x, B x :=
  sorry
end
```

7. Fill in the sorry.

```
variable  U : Type
variables A B C : U → Prop

example (h1 : ∃ x, A x ∧ B x) (h2 : ∀ x, B x → C x) :
    ∃ x, A x ∧ C x :=
sorry
```

8. Complete these proofs.

```
variable  U : Type
variables A B C : U → Prop

example : (¬ ∃ x, A x) → ∀ x, ¬ A x :=
sorry
```

```
example : (∀ x, ¬ A x) → ¬ ∃ x, A x :=
sorry
```

9. Fill in the `sorry`.

```
variable  U : Type
variables R : U → U → Prop

example : (∃ x, ∀ y, R x y) → ∀ y, ∃ x, R x y :=
sorry
```

1. Do the following.

import data.nat open nat

—You can use the facts “odd_succ_of_even” and “odd_mul_of_odd_of_odd”. – Their use is illustrated in the next two examples.

example (x : ℕ) (h1 : even x) : odd (x + 1) := odd_succ_of_even h1

example (x y : ℕ) (h1 : odd x) (h2 : odd y) : odd (x * y) := odd_mul_of_odd_of_odd h1 h2

—Show the following: example : ∀ x y z : ℕ, odd x → odd y → even z → odd ((x * y) * (z + 1)) := sorry

2. The following exercise shows that in the presence of reflexivity, the rules for symmetry and transitivity are equivalent to a single rule.

```
theorem foo {A : Type} {a b c : A} : a = b → c = b → a = c :=
sorry

-- notice that you can now use foo as a rule. The curly braces mean that
-- you do not have to give A, a, b, or c

section
  variable A : Type
  variables a b c : A

  example (h1 : a = b) (h2 : c = b) : a = c :=
  foo h1 h2
end

section
  variable {A : Type}
  variables {a b c : A}

  -- replace the sorry with a proof, using foo and rfl, without using eq.symm.
  theorem my_symm (h : b = a) : a = b :=
  sorry

  -- now use foo, rfl, and my_symm to prove transitivity
  theorem my_trans (h1 : a = b) (h2 : b = c) : a = c :=
  sorry
end
```

3. Replace each "sorry" below by the correct axiom from the list.

```
-- these are the axioms for a commutative ring

#check @add_assoc
#check @add_comm
#check @add_zero
#check @zero_add
#check @mul_assoc
#check @mul_comm
#check @mul_one
#check @one_mul
#check @left_distrib
#check @right_distrib
#check @add_left_neg
#check @add_right_neg
#check @sub_eq_add_neg

variables x y z : int

theorem t1 : x - x = 0 :=
calc
x - x = x + -x : by rw sub_eq_add_neg
    ... = 0      : by rw add_right_neg

theorem t2 (h : x + y = x + z) : y = z :=
calc
y     = 0 + y         : by rw zero_add
    ... = (-x + x) + y : by rw add_left_neg
    ... = -x + (x + y) : by rw add_assoc
    ... = -x + (x + z) : by rw h
    ... = (-x + x) + z : by rw add_assoc
    ... = 0 + z        : by rw add_left_neg
    ... = z            : by rw zero_add

theorem t3 (h : x + y = z + y) : x = z :=
calc
x     = x + 0         : sorry
    ... = x + (y + -y) : sorry
    ... = (x + y) + -y : sorry
    ... = (z + y) + -y : by rw h
    ... = z + (y + -y) : sorry
    ... = z + 0        : sorry
    ... = z            : sorry

theorem t4 (h : x + y = 0) : x = -y :=
calc
x     = x + 0         : by rw add_zero
    ... = x + (y + -y) : by rw add_right_neg
    ... = (x + y) + -y : by rw add_assoc
    ... = 0 + -y       : by rw h
    ... = -y           : by rw zero_add

theorem t5 : x * 0 = 0 :=
have h1 : x * 0 + x * 0 = x * 0 + 0, from
calc
    x * 0 + x * 0 = x * (0 + 0) : sorry
            ... = x * 0       : sorry
            ... = x * 0 + 0   : sorry,
show x * 0 = 0, from t2 _ _ _ h1

theorem t6 : x * (-y) = -(x * y) :=
have h1 : x * (-y) + x * y = 0, from
calc
    x * (-y) + x * y = x * (-y + y) : sorry
                 ... = x * 0        : sorry
                 ... = 0            : by rw t5 x,
show x * (-y) = -(x * y), from t4 _ _ h1

theorem t7 : x + x = 2 * x :=
calc
x + x = 1 * x + 1 * x : by rw one_mul
    ... = (1 + 1) * x   : sorry
    ... = 2 * x         : rfl
```

CHAPTER

TEN

SEMANTICS OF FIRST ORDER LOGIC

In Chapter 6, we emphasized a distinction between the *syntax* and the *semantics* of propositional logic. Syntactic questions have to do with the formal structure of formulas and the conditions under which different types of formulas can be derived. Semantic questions, on the other hand, concern the *truth* of a formula relative to some truth assignment.

As you might expect, we can make a similar distinction in the setting of first order logic. The previous two chapters have focused mainly on syntax, but some semantic ideas have slipped in. Recall the running example with domain of interest $\mathbb{N}$, constant symbols 0, 1, 2, 3, function symbols *add* and *mul*, and predicate symbols *even*, *prime*, *equals*, *lt*, etc. We know that the sentence $\forall y \; \mathit{lt}(0, y)$ is true in this example, if *lt* is interpreted as the less-than relation on the natural numbers. But if we consider the domain $\mathbb{Z}$ instead of $\mathbb{N}$, that same formula becomes false. The sentence $\forall y \; \mathit{lt}(0, y)$ is also false if we consider the domain $\mathbb{N}$, but (somewhat perversely) interpret the predicate $\mathit{lt}(x, y)$ as the relation "x is greater than y" on the natural numbers.

This indicates that the truth or falsity or a first order sentence can depend on how we interpret the quantifiers and basic relations of the language. But some formulas are true under any interpretation: for instance, $\forall y \; (\mathit{lt}(0, y) \to \mathit{lt}(0, y))$ is true of under all the interpretations considered in the last paragraph, and, indeed, under any interpretation we choose. A sentence like this is said to be *valid*; this is the analogue of a tautology in propositional logic, which is true under every possible truth assignment.

We can broaden the analogy: a "model" in first order logic is the analogue of a truth assignment in propositional logic. In the propositional case, choosing a truth assignment allowed us to assign truth values to all formulas of the language; now, choosing an model will allow us to assign truth values to all sentences of a first order language. The aim of the next section is to make this notion more precise.

10.1 Interpretations

The symbols of the language in our running example – 0, 1, *add*, *prime*, and so on – have very indicative names. When we interpret sentences of this language over the domain $\mathbb{N}$, for example, it is clear for which elements of the domain *prime* "should" be true, and for which it "should" be false. But let us consider a first order language that has only two unary predicate symbols *fancy* and *tall*. If we take our domain to be $\mathbb{N}$, is the sentence $\forall x \; (\mathit{fancy}(x) \to \mathit{tall}(x))$ true or false?

The answer, of course, is that we don't have enough information to say. There's no "obvious" meaning to the predicates *fancy* or *tall*, at least not when we apply them to natural numbers. To make sense of the sentence, we need to know which numbers are fancy and which ones are tall. Perhaps multiples of 10 are fancy, and even numbers are tall; in this case, the formula is true, since every multiple of 10 is even. Perhaps prime numbers are fancy and odd numbers are tall; then the formula is false, since 2 is fancy but not tall.

We call each of these descriptions an *interpretation* of the predicate symbols *fancy* and *tall* in the domain $\mathbb{N}$. Formally, an interpretation of a unary predicate P in a domain D is the set of elements of D for which

P is true. For an example, the "standard" interpretation of *prime* in $\mathbb{N}$ that we used above was just the set of prime natural numbers.

We can interpret constant, function, and relation symbols in a similar way. An interpretation of constant symbol c in domain D is an element of D. An interpretation of a function symbol f with arity n is a function that maps n elements of D to another element of D. An interpretation of a relation symbol R with arity n is the set of n tuples of elements of D for which R is true.

It is important to emphasize the difference between a syntactic predicate symbol (or function symbol, or constant symbol) and the semantic predicate (or function, or object) to which it is interpreted. The former is a symbol, relates to other symbols, and has no meaning on its own until we specify an interpretation. Strictly speaking, it makes no sense to write $\mathit{prime}(3)$, where *prime* is a predicate symbol and 3 is a natural number, since the argument to *prime* is supposed to be a syntactic term. Sometimes we may obscure this distinction, as above when we specified a language with constant symbols 0, 1, and 2. But there is still a fundamental distinction between the objects of the domain and the symbols we use to represent them.

Sometimes, when we interpret a language in a particular domain, it is useful to implicitly introduce new constant symbols into the language to denote elements of this domain. Specifically, for each element a of the domain, we introduce a constant symbol $\bar{a}$, which is interpreted as a. Then, the expression $\mathit{prime}(\bar{3})$ *does* make sense. Interpreting the predicate symbol *prime* in the natural way, this expression will evaluate to true. We think of $\bar{3}$ as a linguistic "name" that represents the natural number 3, in the same way that the word "Madonna" is a name that represents the flesh-and-blood pop singer.

10.2 Truth in a Model

Fix a first-order language. Suppose we have chosen a domain D to interpret the language, along with an interpretation in D of each of the symbols of that language. We will call this structure — the domain D, paired with the interpretation — a *model* for the language. A model for a first-order language is directly analogous to a truth assignment for propositional logic, because it provides all the information we need to determine the truth value of each sentence in the language.

The procedure for evaluating the truth of a sentence based on a model works the way you think it should, but the formal description is subtle. Recall the difference between *terms* and *assertions* that we made earlier in Chapter 4. Terms, like a, $x + y$, or $f(c)$, are meant to represent objects. A term does not have a truth value, since (for example) it makes no sense to ask whether 3 is true or false. Assertions, like $P(a)$, $R(x, f(y))$, or $a + b > a \wedge \mathit{prime}(c)$, apply predicate or relation symbols to terms to produce statements that could be true or false.

The interpretation of a term in a model is an element of the domain of that model. The model directly specifies how to interpret constant symbols. To interpret a term $f(t)$ created by applying a function symbol to another term, we interpret the term t, and then apply the interpretation of f to this term. (This process makes sense, since the interpretation of f is a function on the domain.) This generalizes to functions of higher arity in the obvious way. We will not yet interpret terms that include free variables like x and y, since these terms do not pick out unique elements of the domain. (The variable x could potentially refer to any object.)

For example, suppose we have a language with two constant symbols, a and b, a unary function symbol f, and a binary function symbol g. Let $\mathcal{M}$ be the model with domain $\mathbb{N}$, where a and b are interpreted as 3 and 5, respectively, $f(x)$ is interpreted as the function which maps any natural number n to n^2, and g is the addition function. Then the term $g(f(a), b)$ denotes the natural number $3^2 + 5 = 14$.

Similarly, the interpretation of an assertion is a value $\mathbf{T}$ or $\mathbf{F}$. For the sake of brevity, we will introduce new notation here: if A is an assertion and $\mathcal{M}$ is a model of the language of A, we write $\mathcal{M} \models A$ to mean that A evaluates to $\mathbf{T}$ in $\mathcal{M}$, and $\mathcal{M} \not\models A$ to mean that A evaluates to $\mathbf{F}$. (You can read the symbol $\models$ as "satisfies" or "validates.")

To interpret a predicate or relation applied to some terms, we first interpret those terms, and then see if the interpretation of the relation symbol is true of those objects. To continue with the example, suppose our language also has a relation symbol R, and we extend $\mathcal{M}$ to interpret R as the greater-than-or-equal-to relation. Then we have $\mathcal{M} \not\models R(a, b)$, since 3 is not greater than 5, but $\mathcal{M} \models R(g(f(a)), b)$, since 14 is greater than 5.

Interpreting expressions using the logical connectives $\wedge$, $\vee$, $\to$, and $\neg$ works exactly as it did in the propositional setting. $\mathcal{M} \models A \wedge B$ exactly when $\mathcal{M} \models A$ and $\mathcal{M} \models B$, and so on.

We still need to explain how to interpret existential and universal expressions. We saw that $\exists x\ A$ intuitively meant that there was *some* element of the domain that would make A true, when we "replaced" the variable x with that element. To make this a bit more precise, we say that $\mathcal{M} \models \exists x A$ exactly when there is an element a in the domain of $\mathcal{M}$ such that, when we interpret x as a, then $\mathcal{M} \models A$. To continue the example above, we have $\mathcal{M} \models \exists x\ (R(x, b))$, since when we interpret x as 6 we have $\mathcal{M} \models R(x, b)$.

More concisely, we can say that $\mathcal{M} \models \exists x\ A$ when there is an a in the domain of $\mathcal{M}$ such that $\mathcal{M} \models A[\bar{a}/x]$. The notation $A[\bar{a}/x]$ indicates that every occurrence of x in A has been replaced by the symbol $\bar{a}$.

Finally, remember that $\forall x\ A$ meant that A was true for all possible values of x. We make this precise by saying that $\mathcal{M} \models \forall x\ A$ exactly when for every element a in the domain of $\mathcal{M}$, interpreting x as a gives that $\mathcal{M} \models A$. Alternatively, we can say that $\mathcal{M} \models \forall x\ A$ when for every a in the domain of $\mathcal{M}$, we have $\mathcal{M} \models A[\bar{a}/x]$. In our example above, $\mathcal{M} \not\models \forall x\ (R(x, b))$, since when we interpret x as 2 we do not have $\mathcal{M} \models R(x, b)$.

These rules allow us to determine the truth value of any *sentence* in a model. (Remember, a sentence is a formula with no free variables.) There are some subtleties: for instance, we've implicitly assumed that our formula doesn't quantify over the same variable twice, as in $\forall x\ \exists x\ A$. But for the most part, the interpretation process tells us to "read" a formula as talking directly about objects in the domain.

10.3 Examples

Take a simple language with no constant symbols, one relation symbol $\leq$, and one binary function symbol $+$. Our model $\mathcal{M}$ will have domain $\mathbb{N}$, and the symbols will be interpreted as the standard less-than-or-equal-to relation and addition function.

Think about the following questions before you read the answers below. Remember, our domain is $\mathbb{N}$, not $\mathbb{Z}$ or any other number system.

1. Is it true that $\mathcal{M} \models \exists x\ (x \leq x)$? What about $\mathcal{M} \models \forall x\ (x \leq x)$?
2. Similarly, what about $\mathcal{M} \models \exists x\ (x + x \leq x)$? $\mathcal{M} \models \forall x\ (x + x \leq x)$?
3. Do the sentences $\exists x\ \forall y\ (x \leq y)$ and $\forall x\ \exists y\ (x \leq y)$ mean the same thing? Are they true or false?
4. Can you think of a formula A in this language, with one free variable x, such that $\mathcal{M} \models \forall x\ A$ but $\mathcal{M} \not\models \exists x\ A$?

These questions indicate a subtle, and often tricky, interplay between the universal and existential quantifiers. Once you've thought about them a bit, read the answers:

1. Both of these statements are true. For the former, we can (for example) interpret x as the natural number 0. Then, $\mathcal{M} \models x \leq x$, so the existential is true. For the latter, pick an arbitrary natural number n; it is still the case that when we interpret x as n, we have $\mathcal{M} \models x \leq x$.
2. The first statement is true, since we can interpret x as 0. The second statement, though, is false. When we interpret x as 1 (or, in fact, as any natural number besides 0), we see that $\mathcal{M} \not\models x + x \leq x$.
3. These sentences do *not* mean the same thing, although in the specified model, both are true. The first expresses that some natural number is less than or equal to every natural number. This is true: 0 is

less than or equal to every natural number. The second sentence says that for every natural number, there is another natural number at least as big. Again, this is true: every natural number a is less than or equal to a. If we took our domain to be $\mathbb{Z}$ instead of $\mathbb{N}$, the first sentence would be false, while the second would still be true.

4. The situation described here is impossible in our model. If $\mathcal{M} \models \forall x A$, then $\mathcal{M} \models A[\bar{0}/x]$, which implies that $\mathcal{M} \models \exists x A$. The only time this situation can happen is when the domain of our model is empty.

Now consider a different language with constant symbol 2, predicate symbols *prime* and *odd*, and binary relation $<$, interpreted in the natural way over domain $\mathbb{N}$. The sentence $\forall x \; ((2 < x \wedge \mathit{prime}(x)) \to \mathit{odd}(x))$ expresses the fact that every prime number bigger than 2 is odd. It is an example of *relativization*, discussed in Section 7.4. We can now see semantically how relativization works. This sentence is true in our model if, for every natural number n, interpreting x as n makes the sentence true. If we interpret x as 0, 1, or 2, or as any non-prime number, the hypothesis of the implication is false, and thus $(2 < x \wedge \mathit{prime}(x))$ is true. Otherwise, if we interpret x as a prime number bigger than 2, both the hypothesis and conclusion of the implication are true, and $(2 < x \wedge \mathit{prime}(x))$ is again true. Thus the universal statement holds. It was an example like this that partially motivated our semantics for implication back in Chapter 3; any other choice would make relativization impossible.

For the next example, we will consider models that are given by a rectangular grid of "dots." Each dot has a color (red, blue, or green) and a size (small or large). We use the letter R to represent a large red dot and r to represent a small red dot, and similarly for G, g, B, b.

The logical language we use to describe our dot world has predicates *red*, *green*, *blue*, *small* and *large*, which are interpreted in the obvious ways. The relation $\mathit{adj}(x, y)$ is true if the dots referred to by x and y are touching, not on a diagonal. The relations $\mathit{same\text{-}color}(x, y)$, $\mathit{same\text{-}size}(x, y)$, $\mathit{same\text{-}row}(x, y)$, and $\mathit{same\text{-}column}(x, y)$ are also self-explanatory. The relation $\mathit{left\text{-}of}(x, y)$ is true if the dot referred to by x is left of the dot referred to by y, regardless of what rows the dots are in. The interpretations of *right-of*, *above*, and *below* are similar.

Consider the following sentences:

1. $\forall x \; (\mathit{green}(x) \vee \mathit{blue}(x))$
2. $\exists x, y \; (\mathit{adj}(x, y) \wedge \mathit{green}(x) \wedge \mathit{green}(y))$
3. $\exists x \; ((\exists z \mathit{right\text{-}of}(z, x)) \wedge (\forall y (\mathit{left\text{-}of}(x, y) \to \mathit{blue}(y) \vee \mathit{small}(y))))$
4. $\forall x \; (\mathit{large}(x) \to \exists y (\mathit{small}(y) \wedge \mathit{adj}(x, y)))$
5. $\forall x \; (\mathit{green}(x) \to \exists y (\mathit{same\text{-}row}(x, y) \wedge \mathit{blue}(y)))$
6. $\forall x, y \; (\mathit{same\text{-}row}(x, y) \wedge \mathit{same\text{-}column}(x, y) \to x = y)$
7. $\exists x \; \forall y \; (\mathit{adj}(x, y) \to \neg \mathit{same\text{-}size}(x, y))$
8. $\forall x \; \exists y \; (\mathit{adj}(x, y) \wedge \mathit{same\text{-}color}(x, y))$
9. $\exists y \; \forall x \; (\mathit{adj}(x, y) \wedge \mathit{same\text{-}color}(x, y))$
10. $\exists x \; (\mathit{blue}(x) \wedge \exists y (\mathit{green}(y) \wedge \mathit{above}(x, y)))$

We can evaluate them in this particular model:

R	r	g	b
R	b	G	b
B	B	B	b

There they have the following truth values:

1. false

2. true
3. false
4. false
5. true
6. true
7. false
8. true
9. false
10. false

For each sentence, see if you can find a model that makes the sentence true, and another that makes it false. For an extra challenge, try to make all of the sentences true simultaneously. Notice that you can use any number of rows and any number of columns.

10.4 Validity and Logical Consequence

We have seen that whether a formula is true or false often depends on the model we choose. Some formulas, though, are true in every possible model. An example we saw earlier was $\forall y\ (lt(0, y) \to lt(0, y))$. Why is this sentence valid? Suppose $\mathcal{M}$ is an arbitrary model of the language, and suppose a is an arbitrary element of the domain of $\mathcal{M}$. Either $\mathcal{M} \models lt(0, \bar{a})$ or $\mathcal{M} \models \neg lt(0, \bar{a})$. In either case, the propositional semantics of implication guarantee that $\mathcal{M} \models lt(0, \bar{a}) \to lt(0, \bar{a})$. We often write $\models A$ to mean that A is a valid.

In the propositional setting, there is an easy method to figure out if a formula is a tautology or not. Writing the truth table and checking for any rows ending with $\mathbf{F}$ is algorithmic, and we know from the beginning exactly how large the truth table will be. Unfortunately, we cannot do the same for first-order formulas. Any language has infinitely many models, so a "first-order" truth table would be infinitely long. To make matters worse, even checking whether a formula is true in a single model can be a non-algorithmic task. To decide whether a universal statement like $\forall x\ P(x)$ is true in a model with an infinite domain, we might have to check whether P is true of infinitely many elements.

This is not to say that we can *never* figure out if a first-order sentence is a tautology. For example, we have argued that $\forall y\ (lt(0, y) \to lt(0, y))$ was one. It is just a more difficult question than for propositional logic.

As was the case with propositional logic, we can extend the notion of validity to a notion of logical consequence. Fix a first-order language, L. Suppose Γ is a set of sentences in L, and A is a sentence of L. We will say that A *is a logical consequence of* Γ if every model of Γ is a model of A. This is one way of spelling out that A is a "necessary consequence" of A: under any interpretation, if the hypotheses in Γ come out true, A is true as well.

10.5 Soundness and Completeness

In propositional logic, we saw a close connection between the provable formulas and the tautologies – specifically, a formula is provable if and only if it is a tautology. More generally, we say that a formula A is a logical consequence of a set of hypotheses, Γ, if and only if there is a natural deduction proof of A from Γ. It turns out that the analogous statements hold for first order logic.

The "soundness" direction — the fact that if A is provable from Γ then A is true in any model of Γ — holds for reasons that are similar to the reasons it holds in the propositional case. Specifically, the proof proceeds by showing that each rule of natural deduction preserves the truth in a model.

The completeness theorem for first order logic was first proved by Kurt Gödel in his 1929 dissertation. Another, simpler proof was later provided by Leon Henkin.

Theorem. If a formula A is a logical consequence of a set of sentences Γ, then A is provable from Γ.

Compared to the version for propositional logic, the first order completeness theorem is harder to prove. We will not go into too much detail here, but will indicate some of the main ideas. A set of sentences is said to be *consistent* if you cannot prove a contradiction from those hypotheses. Most of the work in Henkin's proof is done by the following "model existence" theorem:

Theorem. Every consistent set of sentences has a model.

From this theorem, it is easy to deduce the completeness theorem. Suppose there is no proof of A from Γ. Then the set $\Gamma \cup \{\neg A\}$ is consistent. (If we could prove $\bot$ from $\Gamma \cup \{\neg A\}$, then by the *reductio ad absurdum* rule we could prove A from Γ.) By the model existence theorem, that means that there is a model $\mathcal{M}$ of $\Gamma \cup \{\neg A\}$. But this is a model of Γ that is not a model of A, which means that A is not a logical consequence of Γ.

The proof of the model existence theorem is intricate. Somehow, from a consistent set of sentences, one has to "build" a model. The strategy is to build the model out of syntactic entities, in other words, to use terms in an expanded language as the elements of the domain.

The moral here is much the same as it was for propositional logic. Because we have developed our syntactic rules with a certain semantics in mind, the two exhibit different sides of the same coin: the provable sentences are exactly the ones that are true in all models, and the sentences that are provable from a set of hypotheses are exactly the ones that are true in all models of those hypotheses.

We therefore have another way to answer the question posed in the previous section. To show that a sentence is a tautology, there is no need to check its proof in every possible model. Rather, it suffices to produce a proof.

10.6 Exercises

1. In a first-order language with a binary relation, $R(x, y)$, consider the following sentences:

   - $\exists x \; \forall y \; R(x, y)$
   - $\exists y \; \forall x \; R(x, y)$
   - $\forall x, y \; (R(x, y) \wedge x \neq y \to \exists z \; (R(x, z) \wedge R(z, y) \wedge x \neq z \wedge y \neq z))$

   For each of the following structures, determine whether of each of those sentences is true or false.

   - the structure $(\mathbb{N}, \leq)$, that is, the interpretation in the natural numbers where R is $\leq$
   - the structure $(\mathbb{Z}, \leq)$
   - the structure $(\mathbb{Q}, \leq)$
   - the structure $(\mathbb{N}, |)$, that is, the interpretation in the natural numbers where R is the "divides" relation
   - the structure $(P(\mathbb{N}), \subseteq)$, that is, the interpretation where variables range over sets of natural numbers, where R is interpreted as the subset relation.

2. Create a 4 x 4 "dots" world that makes all of the following sentences true:
   - $\forall x\ (\mathit{green}(x) \vee \mathit{blue}(x))$
   - $\exists x, y\ (\mathit{adj}(x, y) \wedge \mathit{green}(x) \wedge \mathit{green}(y))$
   - $\exists x\ (\exists z\ \mathit{right\text{-}of}(z, x) \wedge \forall y\ (\mathit{left\text{-}of}(x, y) \to \mathit{blue}(y) \vee \mathit{small}(y)))$
   - $\forall x\ (\mathit{large}(x) \to \exists y\ (\mathit{small}(y) \wedge \mathit{adj}(x, y)))$
   - $\forall x\ (\mathit{green}(x) \to \exists y\ (\mathit{same\text{-}row}(x, y) \wedge \mathit{blue}(y)))$
   - $\forall x, y\ (\mathit{same\text{-}row}(x, y) \wedge \mathit{same\text{-}column}(x, y) \to x = y)$
   - $\exists x\ \forall y\ (\mathit{adj}(x, y) \to \neg \mathit{same\text{-}size}(x, y))$
   - $\forall x\ \exists y\ (\mathit{adj}(x, y) \wedge \mathit{same\text{-}color}(x, y))$
   - $\exists y\ \forall x\ (\mathit{adj}(x, y) \to \mathit{same\text{-}color}(x, y))$
   - $\exists x\ (\mathit{blue}(x) \wedge \exists y\ (\mathit{green}(y) \wedge \mathit{above}(x, y)))$
3. Fix a first-order language L, and let A and B be any two sentences in L. Remember that $\vDash A$ means that A is valid. Unpacking the definition, show that if $\vDash A \wedge B$, then $\vDash A$ and $\vDash B$.
4. Give a concrete example to show that $\vDash A \vee B$ does not necessarily imply $\vDash A$ or $\vDash B$. In other words, pick a language L and choose particular sentences A and B such that $A \vee B$ is valid, but neither A nor B is valid.

CHAPTER

ELEVEN

SETS

We have come to a turning point in this textbook. We will henceforth abandon natural deduction, for the most part, and focus on ordinary mathematical proofs. We will continue to think about how informal mathematics can be represented in symbolic terms, and how the rules of natural deduction play out in the informal setting. But the emphasis will be on writing ordinary mathematical arguments, not designing proof trees. Lean will continue to serve as a bridge between the informal and formal realms.

In this chapter, we consider a notion that has come to play a fundamental role in mathematical reasoning, namely, that of a "set."

11.1 Elementary Set Theory

In a publication in the journal *Mathematische Annalen* in 1895, the German mathematician Georg Cantor presented the following characterization of the notion of a *set* (or *Menge*, in his terminology):

> By a *set* we mean any collection M of determinate, distinct objects (called the *elements* of M) of our intuition or thought into a whole.

Since then, the notion of a set has been used to unify a wide range of abstractions and constructions. Axiomatic set theory, which we will discuss in a later chapter, provides a foundation for mathematics in which everything can be viewed as a set.

On a broad construal, *any* collection can be a set; for example, we can consider the set whose elements are Ringo Star, the number 7, and the set whose only member is the Empire State Building. With such a broad notion of set we have to be careful: Russell's paradox has us consider the set S of all sets that are not elements of themselves, which leads to a contradiction when we ask whether S is an element of itself. (Try it!) The axioms of set theory tell us what sets exist, and have been carefully designed to avoid paradoxical sets like that of the Russell paradox.

In practice, mathematicians are not so freewheeling in their use of sets. Typically, one fixes a domain such as the natural numbers, and consider subsets of that domain. In other words, we consider sets of numbers, sets of points, sets of lines, and so on, rather than arbitrary "sets." In this text, we will adopt this convention: when we talk about sets, we are always implicitly talking about sets of elements of some domain.

Given a set A of objects in some domain and an object x, we write $x \in A$ to say that x is an element of A. Cantor's characterization suggests that whenever we have some property, P, of a domain, we can form the set of elements that have that property. This is denoted using "set-builder notation" as $\{x \mid P(x)\}$. For example, we can consider all the following sets of natural numbers:

- $\{n \mid n \text{ is even}\}$
- $\{n \mid n \text{ is prime}\}$
- $\{n \mid n \text{ is prime and greater than } 2\}$

- $\{n \mid n$ can be written as a sum of squares$\}$
- $\{n \mid n$ is equal to 1, 2, or 3$\}$

This last set is written more simply $\{1, 2, 3\}$. If the domain is not clear from the context, we can specify it by writing it explicitly, for example, in the expression $\{n \in \mathbb{N} \mid n \text{ is even}\}$.

Using set-builder notation, we can define a number of common sets and operations. The *empty set*, $\emptyset$, is the set with no elements:

$$\emptyset = \{x \mid \text{false}\}$$

Dually, we can define the *universal set*, $\mathcal{U}$, to be the set consisting of every element of the domain:

$$\mathcal{U} = \{x \mid \text{true}\}$$

Given two sets A and B, we define their *union* to be the set of elements in either one:

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

And we define their *intersection* to be the set of elements of both:

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

We define the *complement* of a set of A to be the set of elements that are not in A:

$$\overline{A} = \{x \mid x \notin A\}$$

We define the *set difference* of two sets A and B to be the set of elements in A but not B:

$$A \setminus B = \{x \mid x \in A \text{ and } x \notin B\}$$

Two sets are said to be equal if they have exactly the same elements. If A and B are sets, A is said to be a *subset* of B, written $A \subseteq B$, if every element of A is an element of B. Notice that A is equal to B if and only if A is a subset of B and B is a subset of A.

Notice also that just everything we have said about sets so far is readily representable in symbolic logic. We can render the defining properties of the basic sets and constructors as follows:

$$\begin{aligned}
&\forall x\ (x \in \emptyset \leftrightarrow \bot)\\
&\forall x\ (x \in \mathcal{U} \leftrightarrow \top)\\
&\forall x\ (x \in A \cup B \leftrightarrow x \in A \vee x \in B)\\
&\forall x\ (x \in A \cap B \leftrightarrow x \in A \wedge x \in B)\\
&\forall x\ (x \in \overline{A} \leftrightarrow x \notin A)\\
&\forall x\ (x \in A \setminus B \leftrightarrow x \in A \wedge x \notin B)
\end{aligned}$$

The assertion that A is a subset of B can be written $\forall x\ (x \in A \to x \in B)$, and the assertion that A is equal to be can be written $\forall x\ (x \in A \leftrightarrow x \in B)$. These are all *universal* statements, that is, statements with universal quantifiers in front, followed by basic assertions and propositional connectives. What this means is that reasoning about sets formally often amounts to using nothing more than the rules for the universal quantifier together with the rules for propositional logic.

Logicians sometimes describe ordinary mathematical proofs as *informal*, in contrast to the *formal proofs* in natural deduction. When writing informal proofs, the focus is on readability. Here is an example.

Theorem. Let A, B, and C denote sets of elements of some domain, $\mathcal{U}$. Then $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$.

Proof. Let x be arbitrary, and suppose x is in $A \cap (B \cup C)$. Then x is in A, and either x is in B or x is in C. In the first case, x is in A and B, and hence in $A \cap B$. In the second case, x is in A and C, and hence $A \cap C$. Either way, we have that x is in $(A \cap B) \cup (A \cap C)$.

Conversely, suppose x is in $(A \cap B) \cup (A \cap C)$. There are now two cases.

First, suppose x is in $A \cap B$. Then x is in both A and B. Since x is in B, it is also in $B \cup C$, and so x is in $A \cap (B \cup C)$.

The second case is similar: suppose x is in $A \cap C$. Then x is in both A and C, and so also in $B \cup C$. Hence, in this case also, x is in $A \cap (B \cup C)$, as required.

Notice that this proof does not look anything like a proof in symbolic logic. For one thing, ordinary proofs tend to favor words over symbols. Of course, mathematics uses symbols all the time, but not in place of words like "and" and "not"; you will rarely, if ever, see the symbols $\wedge$ and $\neg$ in a mathematics textbook, unless it is a textbook specifically about logic.

Similarly, the structure of an informal proof is conveyed with ordinary paragraphs and punctuation. Don't rely on pictorial diagrams, line breaks, and indentation to convey the structure of a proof. Rather, you should rely on literary devices like signposting and foreshadowing. It is often helpful to present an outline of a proof or the key ideas before delving into the details, and the introductory sentence of a paragraph can help guide a reader's expectations, just as it does in an expository essay.

Nonetheless, you should be able to see elements of natural deduction implicitly in the proof above. In formal terms, the theorem is equivalent to the assertion

$$\forall x \; (x \in A \cap (B \cup C) \leftrightarrow x \in (A \cap B) \cup (A \cap C)),$$

and the proof proceeds accordingly. The phrase "let x be arbitrary" is code for the $\forall$ introduction rule, and the form of the rest of the proof is a $\leftrightarrow$ introduction. Saying that x is in $A \cap (B \cup C)$ is implicitly an "and," and the argument uses $\wedge$ elimination to get $x \in A$ and $x \in B \cup C$. Saying $x \in B \cup C$ is implicitly an "or," and the proof then splits on cases, depending on whether $x \in B$ or $x \in C$.

Modulo the unfolding of definition of intersection and union in terms of "and" and "or," the "only if" direction of the previous proof could be represented in natural deduction like this:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{y \in A \cap (B \cup C)}{}^{1}}{y \in B \cup C} \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{y \in A \cap (B \cup C)}{}^{1}}{y \in A} \qquad \dfrac{}{y \in B}{}^{2}}{y \in A \cap B}}{y \in (A \cap B) \cup (A \cap C)} \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{y \in A \cap (B \cup C)}{}^{1}}{y \in A} \qquad \dfrac{}{y \in C}{}^{2}}{y \in A \cap C}}{y \in (A \cap B) \cup (A \cap C)}}{y \in (A \cap B) \cup (A \cap C)}{}^{2} \qquad \vdots}{\dfrac{y \in A \cap (B \cup C) \leftrightarrow y \in (A \cap B) \cup (A \cap C)}{\forall x \; (x \in A \cap (B \cup C) \leftrightarrow x \in (A \cap B) \cup (A \cap C))}}{}^{1}$$

In the next chapter, we will see that this logical structure is made manifest in Lean. But writing long proofs in natural deduction is not the most effective to communicate the mathematical ideas. So our goal here is to teach you to think in terms of natural deduction rules, but express the steps in ordinary English.

Here is another example.

Theorem. $(A \setminus B) \setminus C = A \setminus (B \cup C)$.

Proof. Let x be arbitrary, and suppose x is in $(A \setminus B) \setminus C$. Then x is in $A \setminus B$ but not C, and hence it is in A but not in B or C. This means that x is in A but not $B \cup C$, and so in $A \setminus (B \cup C)$.

Conversely, suppose x is in $A \setminus (B \cup C)$. Then x is in A, but not in $B \cup C$. In particular, x is in neither B nor C, because otherwise it would be in $B \cup C$. So x is in $A \setminus B$, and hence in $(A \setminus B) \setminus C$.

Perhaps the biggest difference between informal proofs and formal proofs is the level of detail. Informal proofs will often skip over details that are taken to be "straightforward" or "obvious," devoting more effort to spelling out inferences that are novel of unexpected.

Writing a good proof is like writing a good essay. To convince your readers that the conclusion is correct, you have to get them to understand the argument, without overwhelming them with unnecessary details. It helps to have a specific audience in mind. Try speaking the argument aloud to friends, roommates, and family members; if their eyes glaze over, it is unreasonable to expect anonymous readers to do better.

One of the best ways to learn to write good proofs is to *read* good proofs, and pay attention to the style of writing. Pick an example of a textbook that you find especially clear and engaging, and think about what makes it so.

Natural deduction and formal verification can help you understand the components that make a proof *correct*, but you will have to develop an intuitive feel for what makes a proof easy and enjoyable to read.

11.2 Calculations with Sets

Calculation is a central to mathematics, and mathematical proofs often involve carrying out a sequence of calculations. Indeed, a calculation can be viewed as a proof in and of itself that two expressions describe the same entity.

In high school algebra, students are often asked to prove identities like the following:

Proposition. $\frac{n(n+1)}{2} + (n+1) = \frac{(n+1)(n+2)}{2}$, for every natural number n.

In some places, students are asked to write proofs like this:

Proof.

$$\begin{aligned} \frac{n(n+1)}{2} + (n+1) &=? \frac{(n+1)(n+2)}{2} \\ \frac{n^2+n}{2} + \frac{2n+2}{2} &=? \frac{n^2+3n+2}{2} \\ \frac{n^2+n+2n+2}{2} &=? \frac{n^2+3n+2}{2} \\ \frac{n^2+3n+2}{2} &= \frac{n^2+3n+2}{2} \end{aligned}$$

Mathematicians generally cringe when they see this. *Don't do it!* It looks like an instance of forward reasoning, where we start with a complex identity and end up proving $x = x$. Of course, what is really meant is that each line follows from the next. There is a way of expressing this, with the phrase "it suffices to show." The following presentation comes closer to mathematical vernacular:

Proof. We want to show

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

To do that, it suffices to show

$$\frac{n^2+n}{2} + \frac{2n+2}{2} = \frac{n^2+3n+2}{2}.$$

For that, it suffices to show

$$\frac{n^2+n+2n+2}{2} = \frac{n^2+3n+2}{2}.$$

But this last equation is clearly true.

The narrative doesn't flow well, however. Sometimes there are good reasons to work backwards in a proof, but in this case it is easy to present the proof in a more forward-directed manner. Here is one example:

Proof. Calculating on the left-hand side, we have

$$\begin{aligned}\frac{n(n+1)}{2} + (n+1) &= \frac{n^2+n}{2} + \frac{2n+2}{2}\\ &= \frac{n^2+n+2n+2}{2}\\ &= \frac{n^2+3n+2}{2}.\end{aligned}$$

On the right-hand side, we also have

$$\frac{(n+1)(n+2)}{2} = \frac{n^2+3n+2}{2}.$$

So $\frac{n(n+1)}{2} + (n+1) = \frac{n^2+3n+2}{2}$, as required.

Mathematicians often use the abbreviations "LHS" and "RHS" for "left-hand side" and "right-hand side," respectively, in situations like this. In fact, here we can easily write the proof as a single forward-directed calculation:

Proof.

$$\begin{aligned}\frac{n(n+1)}{2} + (n+1) &= \frac{n^2+n}{2} + \frac{2n+2}{2}\\ &= \frac{n^2+n+2n+2}{2}\\ &= \frac{n^2+3n+2}{2}\\ &= \frac{(n+1)(n+2)}{2}.\end{aligned}$$

Such a proof is clear, compact, and easy to read. The main challenge to the reader is to figure out what justifies each subsequent step. Mathematicians sometimes annotate such a calculation with additional information, or add a few words of explanation in the text before and/or after. But the ideal situation is to carry out the calculation is small enough steps so that each step is straightforward, and needs to no explanation. (And, once again, what counts as "straightforward" will vary depending on who is reading the proof.)

We have said that two sets are equal if they have the same elements. In the previous section, we proved that two sets are equal by reasoning about the elements of each, but we can often be more efficient. Assuming A, B, and C are subsets of some domain $\mathcal{U}$, the following identities hold:

- $A \cup \overline{A} = \mathcal{U}$
- $A \cap \overline{A} = \emptyset$
- $\overline{\overline{A}} = A$
- $A \cup A = A$
- $A \cap A = A$
- $A \cup \emptyset = A$
- $A \cap \emptyset = \emptyset$
- $A \cup \mathcal{U} = \mathcal{U}$
- $A \cap \mathcal{U} = A$
- $A \cup B = B \cup A$
- $A \cap B = B \cap A$
- $(A \cup B) \cup C = A \cup (B \cup C)$
- $(A \cap B) \cap C = A \cap (B \cap C)$
- $\overline{A \cap B} = \overline{A} \cup \overline{B}$
- $\overline{A \cup B} = \overline{A} \cap \overline{B}$
- $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$
- $A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$
- $A \cap (A \cup B) = A$
- $A \cup (A \cap B) = A$

This allows us to prove further identities by calculating. Here is an example.

Theorem. Let A and B be subsets of some domain $\mathcal{U}$. Then $(A \cap \overline{B}) \cup B = A \cup B$.

Proof.

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

Here is another example.

Theorem. Let A and B be subsets of some domain $\mathcal{U}$. Then $(A \setminus B) \cup (B \setminus A) = (A \cup B) \setminus (A \cap B)$.

Proof.

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

Classically, you may have noticed that propositions, under logical equivalence, satisfy identities similar to sets. That is no coincidence; both are instances of *boolean algebras*. Here are the identities above translated to the language of a boolean algebra:

- $A \vee \neg A = \top$
- $A \wedge \neg A = \bot$
- $\neg\neg A = A$
- $A \vee A = A$
- $A \wedge A = A$
- $A \vee \bot = A$
- $A \wedge \bot = \bot$
- $A \vee \top = \top$
- $A \wedge \top = A$
- $A \vee B = B \vee A$
- $A \wedge B = B \wedge A$
- $(A \vee B) \vee C = A \vee (B \vee C)$
- $(A \wedge B) \wedge C = A \wedge (B \wedge C)$
- $\neg A \wedge B = \neg A \vee \neg B$
- $\neg A \vee B = \neg A \wedge \neg B$
- $A \wedge (B \vee C) = (A \wedge B) \vee (A \wedge C)$
- $A \vee (B \wedge C) = (A \vee B) \wedge (A \vee C)$
- $A \wedge (A \vee B) = A$
- $A \vee (A \wedge B) = A$

Translated to propositions, the first theorem above is as follows:

Theorem. Let A and B be elements of a boolean algebra. Then $(A \wedge \neg B) \vee B = B$.

Proof.

$$\begin{aligned}(A \wedge \neg B) \vee B &= (A \vee B) \wedge (\neg B \vee B) \\ &= (A \vee B) \wedge \top \\ &= (A \vee B).\end{aligned}$$

11.3 Indexed Families of Sets

If I is a set, we will sometimes wish to consider a *family* $(A_i)_{i \in I}$ of sets indexed by elements of I. For example, we might be interested in a sequence

$$A_0, A_1, A_2, \ldots$$

of sets indexed by the natural numbers. The concept is best illustrated by some examples.

- For each natural number n, we can define the set A_n to be the set of people alive today that are of age n. For each age we have the corresponding set. Someone of age 20 is an element of the set A_{20}, while a newborn baby is an element of A_0. The set A_{200} is empty. This family $(A_n)_{n \in \mathbb{N}}$ is a is a family of sets indexed by the natural numbers.
- For every real number r we can define B_r to be the set of positive real numbers larger than r, so $B_r = \{x \in \mathbb{R} \mid x > r \text{ and } x > 0\}$. Then $(B_r)_{r \in \mathbb{R}}$ is a family of sets indexed by the real numbers.
- For every natural number n we can define $C_n = \{k \in \mathbb{N} \mid k \text{ is a divisor of } n\}$ as the set of divisors of n.

Given a family $(A_i)_{i \in I}$ of sets indexed by I, we can form its *union*:

$$\bigcup_{i \in I} A_i = \{x \mid x \in A_i \text{ for some } i \in I\}$$

We can also form the *intersection* of a family of sets:

$$\bigcap_{i \in I} A_i = \{x \mid x \in A_i \text{ for every } i \in I\}$$

So an element x is in $\bigcup_{i \in I} A_i$ if and only if x is in A_i for *some* i in I, and x is in $\bigcap_{i \in I} A_i$ if and only if x is in A_i for every i in I. These operations are represented in symbolic logic by the existential and the universal quantifiers. We have:

- $\forall x \, (x \in \bigcup_{i \in I} A_i \leftrightarrow \exists i \in I \, (x \in A_i))$
- $\forall x \, (x \in \bigcap_{i \in I} A_i \leftrightarrow \forall i \in I \, (x \in A_i))$

Returning to the examples above, we can compute the union and intersection of each family. For the first example, $\bigcup_{n \in \mathbb{N}} A_n$ is the set of all living people, and $\bigcap_{n \in \mathbb{N}} A_n = \emptyset$. Also, $\bigcup_{r \in \mathbb{R}} B_r = \mathbb{R}_{>0}$, the set of all positive real numbers, and $\bigcap_{r \in \mathbb{R}} B_r = \emptyset$. For the last example, we have $\bigcup_{n \in \mathbb{N}} C_n = \mathbb{N}$ and $\bigcap_{n \in \mathbb{N}} C_n = \{1\}$, since 1 is a divisor of every natural number.

Suppose that I contains just two elements, say $I = \{c, d\}$. Let $(A_i)_{i \in I}$ be a family of sets indexed by I. Because I has two elements, this family consists of just the two sets A_c and A_d. Then the union and intersection of this family are just the union and intersection of the two sets:

$$\bigcup_{i \in I} A_i = A_c \cup A_d$$
$$\bigcap_{i \in I} A_i = A_c \cap A_d.$$

This means that the union and intersection of two sets are just a special case of the union and intersection of a family of sets.

We also have equalities for unions and intersections of families of sets. Here are a few of them:

- $A \cap \bigcup_{i \in I} B_i = \bigcup_{i \in I} (A \cap B_i)$
- $A \cup \bigcap_{i \in I} B_i = \bigcap_{i \in I} (A \cup B_i)$
- $\overline{\bigcap_{i \in I} A_i} = \bigcup_{i \in I} \overline{A_i}$
- $\overline{\bigcup_{i \in I} A_i} = \bigcap_{i \in I} \overline{A_i}$
- $\bigcup_{i \in I} \bigcup_{j \in J} A_{i,j} = \bigcup_{j \in J} \bigcup_{i \in I} A_{i,j}$
- $\bigcap_{i \in I} \bigcap_{j \in J} A_{i,j} = \bigcap_{j \in J} \bigcap_{i \in I} A_{i,j}$

In the last two lines, $A_{i,j}$ is indexed by two sets I and J. This means that for every $i \in I$ and $j \in J$ we have a set $A_{i,j}$. For the first four equalities, try to figure out what the rule means if the index set I contains two elements.

Let's prove the first identity. Notice how the logical forms of the assertions $x \in A \cap \bigcup_{i \in I} B_i$ and $x \in \bigcup_{i \in I} (A \cap B_i)$ dictate the structure of the proof.

Theorem. Let A be any subset of some domain U, and let $(B_i)_{i \in I}$ be a family of subsets of U indexed by I. Then

$$A \cap \bigcup_{i \in I} B_i = \bigcup_{i \in I} (A \cap B_i)$$

Proof. Suppose x is in $A \cap \bigcup_{i \in I} B_i$. Then x is in A and x is in B_j for some $j \in I$. So x is in $A \cap B_j$, and hence in $\bigcup_{i \in I} (A \cap B_i)$.

Conversely, suppose x is in $\bigcup_{i \in I} (A \cap B_i)$. Then, for some j in I, x is in $A \cap B_j$. Hence x is in A, and since x is in B_j, it is in $\bigcup_{i \in I} B_i$. Hence x is in $A \cap \bigcup_{i \in I} B_i$, as required.

11.4 Cartesian Product and Power Set

The *ordered pair* of two objects a and b is denoted (a, b). We say that a is the *first component* and b is the *second component* of the pair. Two pairs are only equal if the first component are equal and the second components are equal. In symbols, $(a, b) = (c, d)$ if and only if $a = c$ and $b = d$.

Some axiomatic foundations take the notion of a pair to be primitive. In axiomatic set theory, it is common to *define* an ordered pair to be a particular set, namely

$$(a, b) = \{\{a\}, \{a, b\}\}.$$

Notice that if $a = b$, this set has only one element:

$$(a, a) = \{\{a\}, \{a, a\}\} = \{\{a\}, \{a\}\} = \{\{a\}\}.$$

The following theorem shows that this definition is reasonable.

Theorem. Using the definition of ordered pairs above, we have $(a, b) = (c, d)$ if and only if $a = c$ and $b = d$.

Proof. If $a = c$ and $b = d$ then clearly $(a, b) = (c, d)$. For the other direction, suppose that $(a, b) = (c, d)$, which means

$$\underbrace{\{\{a\}, \{a, b\}\}}_{L} = \underbrace{\{\{c\}, \{c, d\}\}}_{R}.$$

Suppose first that $a = b$. Then $L = \{\{a\}\}$. This means that $\{c\} = \{a\}$ and $\{c, d\} = \{a\}$, from which we conclude that $c = a$ and $d = a = b$.

Now suppose that $a \neq b$. If $\{c\} = \{a, b\}$ then we conclude that a and b are both equal to c, contradicting $a \neq b$. Since $\{c\} \in L$, $\{c\}$ must be equal to $\{a\}$, which means that $a = c$. We know that $\{a, b\} \in R$, and since we know $\{a, b\} \neq \{c\}$, we conclude $\{a, b\} = \{c, d\}$. This means that $b \in \{c, d\}$, since $b \neq a = c$, we conclude that $b = d$.

Hence in both cases we conclude that $a = c$ and $b = d$, proving the theorem.

Using ordered pairs we can define the *ordered triple* (a, b, c) to be $(a, (b, c))$. Then we can prove that $(a, b, c) = (d, e, f)$ if and only if $a = d$, $b = e$ and $c = f$, which you are asked to do in the exercises. We can also define ordered n-tuples, which are sequence of n objects, in a similar way.

Given two sets A and B, we define the *cartesian product* $A \times B$ of these two sets as the set of all pairs where the first component is an element in A and the second component is an element in B. In set-builder notation this means

$$A \times B = \{(a, b) \ \mid a \ \in A \text{ and } b \in B\}.$$

Note that if A and B are subsets of a particular domain $\mathcal{U}$, the set $A \times B$ need not be a subset of the same domain. However, it will be a subset of $\mathcal{U} \times \mathcal{U}$.

Given a set A we can define the *power set* $\mathcal{P}(A)$ to be the set of all subsets of A. In set-builder notation we can write this as

$$\mathcal{P}(A) = \{B \mid B \subseteq A\}.$$

If A is a subset of $\mathcal{U}$, $\mathcal{P}(A)$ may not be a subset $\mathcal{U}$, but it is always a subset of $\mathcal{P}(\mathcal{U})$.

11.5 Exercises

1. Prove the following theorem: Let A, B, and C be sets of elements of some domain. Then $A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$. (Henceforth, if we don't specify natural deduction or Lean, "prove" and "show" mean give an ordinary mathematical proof, using ordinary mathematical language rather than symbolic logic.)
2. Prove the following theorem: Let A and B be sets of elements of some domain. Then $\overline{A \setminus B} = \overline{A} \cup B$.
3. Two sets A and B are said to be *disjoint* if they have no element in common. Show that if A and B are disjoint, $C \subseteq A$, and $D \subseteq B$, then C and D are disjoint.
4. Let A and B be sets. Show $(A \setminus B) \cup (B \setminus A) = (A \cup B) \setminus (A \cap B)$, by showing that both sides have the same elements.
5. Let A, B, and C be subsets of some domain $\mathcal{U}$. Give a calculational proof of the identity $A \setminus (B \cup C) = (A \setminus B) \setminus C$, using the identities above. Also use the fact that, in general, $C \setminus D = C \cap \overline{D}$.
6. Similarly, give a calculational proof of $(A \setminus B) \cup (A \cap B) = A$.
7. Give calculational proofs of the following:
   - $A \setminus B = A \setminus (A \cap B)$
   - $A \setminus B = (A \cup B) \setminus B$
   - $(A \cap B) \setminus C = (A \setminus C) \cap B$

8. Prove that if $(A_{i,j})_{i \in I, j \in J}$ is a family indexed by two sets I and J, then

$$\bigcup_{i \in I} \bigcap_{j \in J} A_{i,j} \subseteq \bigcap_{j \in J} \bigcup_{i \in I} A_{i,j}.$$

   Also, find a family $(A_{i,j})_{i \in I, j \in J}$ where the reverse inclusion does not hold.

9. Prove using calculational reasoning that

$$\left(\bigcup_{i \in I} A_i \right) \cap \left(\bigcup_{j \in J} B_j \right) = \bigcup_{\substack{i \in I \\ j \in J}} (A_i \cap B_j).$$

   The notation $\bigcup_{\substack{i \in I \\ j \in J}} (A_i \cap B_j)$ means $\bigcup_{i \in I} \bigcup_{j \in J} (A_i \cap B_j)$.

10. Using the definition $(a, b, c) = (a, (b, c))$, show that $(a, b, c) = (d, e, f)$ if and only if $a = d$, $b = e$ and $c = f$.

11. Prove that $A \times (B \cup C) = (A \times B) \cup (A \times C)$

12. Prove that $(A \cap B) \times (C \cap D) = (A \times C) \cap (B \times D)$. Find an expression for $(A \cup B) \times (C \cup D)$ consisting of unions of cartesian products, and prove that your expression is correct.

13. Prove that that $A \subseteq B$ if and only if $\mathcal{P}(A) \subseteq \mathcal{P}(B)$.

CHAPTER

TWELVE

SETS IN LEAN

In the last chapter, we noted that although in axiomatic set theory one consider sets of disparate objects, it is more common in mathematics to consider subsets of some fixed domain, $\mathcal{U}$. This is the way sets are handled in Lean. For any data type `U`, Lean gives us a new data type, `set U`, consisting of the sets of elements of `U`. Thus, for example, we can reason about sets of natural numbers, or sets of integers, or sets of pairs of natural numbers.

12.1 Basics

Given `A : set U` and `x : U`, we can write `x ∈ A` to state that `x` is a member of the set `A`. The character ∈ can be typed using `\in`. We need to import the library file `data.set` and open the "namespace" set to have the notions and notations made available to us.

```
import data.set
open set

variable {U : Type}
variables A B C : set U
variable x : U

#check x ∈ A
#check A ∪ B
#check B \ C
#check C ∩ A
#check -C
#check ∅ ⊆ A
#check B ⊆ univ
```

You can type the symbols ⊆, ∅, ∪, ∩, \ as `\subeq` `\empty`, `\un`, `\i`, and `\\`, respectively. We have made the type variable `U` implicit, because it can typically be inferred from context. The universal set is denoted `univ`, and set complementation is denoted with a negation symbol.

The following pattern can be used to show that `A` is a subset of `B`:

```
example : A ⊆ B :=
assume x,
assume h : x ∈ A,
show x ∈ B, from sorry
```

And the following pattern be used to show that `A` and `B` are equal:

```
example : A = B :=
eq_of_subset_of_subset
```

```
  (assume x,
    assume h : x ∈ A,
    show x ∈ B, from sorry)
  (assume x,
    assume h : x ∈ B,
    show x ∈ A, from sorry)
```

Alternatively, we can use the following pattern:

```
example : A = B :=
ext (assume x, iff.intro
  (assume h : x ∈ A,
    show x ∈ B, from sorry)
  (assume h : x ∈ B,
    show x ∈ A, from sorry))
```

Here, `ext` is short for "extensionality." In symbolic terms, it is the following fact:

$$\forall x\ (x \in A \leftrightarrow x \in B) \to A = B.$$

This reduces proving $A = B$ to proving $\forall x\ (x \in A \leftrightarrow x \in B)$, which we can do using $\forall$ and $\leftrightarrow$ introduction.

Moreover, Lean supports the following nifty feature: the defining rules for union, intersection and other operations on sets are considered to hold "definitionally." This means that the expressions `x ∈ A ∩ B` and `x ∈ A ∧ x ∈ B` mean the same thing to Lean. This is the same for the other constructions on sets; for example `x ∈ A \ B` and `x ∈ A ∧ ¬ (x ∈ B)` mean the same thing to Lean. You can also write `x ∉ B` for `¬ (x ∈ B)`, where ∉ is written using `\notin`. For the other set constructions, the defining equivalences in the last chapter hold definitionally. The following example illustrates these features.

```
example : ∀ x, x ∈ A → x ∈ B → x ∈ A ∩ B :=
assume x,
assume : x ∈ A,
assume : x ∈ B,
show x ∈ A ∩ B, from and.intro ‹x ∈ A› ‹x ∈ B›

example : A ⊆ A ∪ B :=
assume x,
assume : x ∈ A,
show x ∈ A ∪ B, from or.inl this

example : ∅ ⊆ A  :=
assume x,
assume : x ∈ ∅,
show x ∈ A, from false.elim ‹x ∈ (∅ : set U)›
```

Remember from Section 4.5 that we can use `assume` without a label, and refer back to hypotheses using French quotes. We have used this feature in the previous example. Without that feature, we could have written the examples above as follows:

```
example : ∀ x, x ∈ A → x ∈ B → x ∈ A ∩ B :=
assume x,
assume h1 : x ∈ A,
assume h2 : x ∈ B,
show x ∈ A ∩ B, from and.intro h1 h2

example : A ⊆ A ∪ B :=
assume x,
assume h : x ∈ A,
show x ∈ A ∪ B, from or.inl h

example : ∅ ⊆ A  :=
assume x,
assume : x ∈ ∅,
show x ∈ A, from false.elim ‹x ∈ (∅ : set U)›
```

Below, and in the chapters that follow, we will begin to use `assume` and `have` command without labels.

Notice also that in the last example, we had to annotate the empty set by writing `(∅ : set U)` to tell Lean which empty set we mean. Lean can often infer information like this from the context (for example, from the fact that we are trying to show `x ∈ A`, where `A` has type `set U`), but in this case, it needs a bit more help.

Alternatively, we can use versions theorems in the Lean library that are designed specifically for use with sets:

```
example : ∀ x, x ∈ A → x ∈ B → x ∈ A ∩ B :=
assume x,
assume : x ∈ A,
assume : x ∈ B,
show x ∈ A ∩ B, from mem_inter ‹x ∈ A› ‹x ∈ B›

example : A ⊆ A ∪ B :=
assume x,
assume h : x ∈ A,
show x ∈ A ∪ B, from mem_union_left B h

example : ∅ ⊆ A  :=
assume x,
assume : x ∈ ∅,
show x ∈ A, from absurd this (not_mem_empty x)
```

Remember that `absurd` can be used to prove any fact from two contradictory hypotheses `h1 : P` and `h2 : ¬ P`. Here the `not_mem_empty x` is the fact `x ∉ ∅`. You can see the statements of the theorems using the `#check` command in Lean:

```
#check @mem_inter
#check @mem_of_mem_inter_left
#check @mem_of_mem_inter_right
#check @mem_union_left
#check @mem_union_right
#check @mem_or_mem_of_mem_union
#check @not_mem_empty
```

Here, the `@` symbol in Lean prevents it from trying to fill in implicit arguments automatically, and instead display the full statement of the theorem.

The fact that Lean can identify sets with their logical definitions makes it easy to prove inclusions between sets:

```
example : A \ B ⊆ A :=
assume x,
assume : x ∈ A \ B,
show x ∈ A, from and.left this

example : A \ B ⊆ -B :=
assume x,
assume : x ∈ A \ B,
have x ∉ B, from and.right this,
show x ∈ -B, from this
```

Once again, we can use versions of the theorems designed specifically for sets:

```
example : A \ B ⊆ A :=
assume x,
assume : x ∈ A \ B,
show x ∈ A, from mem_of_mem_diff this

example : A \ B ⊆ -B :=
assume x,
assume : x ∈ A \ B,
have x ∉ B, from not_mem_of_mem_diff this,
show x ∈ -B, from this
```

12.2 Some Identities

here is the proof of the first identity that we proved informally in the previous chapter:

```
example : A ∩ (B ∪ C) = (A ∩ B) ∪ (A ∩ C) :=
eq_of_subset_of_subset
  (assume x,
    assume h : x ∈ A ∩ (B ∪ C),
    have x ∈ A, from and.left h,
    have x ∈ B ∪ C, from and.right h,
    or.elim (‹x ∈ B ∪ C›)
      (assume : x ∈ B,
        have x ∈ A ∩ B, from and.intro ‹x ∈ A› ‹x ∈ B›,
        show x ∈ (A ∩ B) ∪ (A ∩ C), from or.inl this)
      (assume : x ∈ C,
        have x ∈ A ∩ C, from and.intro ‹x ∈ A› ‹x ∈ C›,
        show x ∈ (A ∩ B) ∪ (A ∩ C), from or.inr this))
  (assume x,
    assume : x ∈ (A ∩ B) ∪ (A ∩ C),
    or.elim this
      (assume h : x ∈ A ∩ B,
        have x ∈ A, from and.left h,
        have x ∈ B, from and.right h,
        have x ∈ B ∪ C, from or.inl this,
        show x ∈ A ∩ (B ∪ C), from and.intro ‹x ∈ A› this)
      (assume h : x ∈ A ∩ C,
        have x ∈ A, from and.left h,
        have x ∈ C, from and.right h,
        have x ∈ B ∪ C, from or.inr this,
        show x ∈ A ∩ (B ∪ C), from and.intro ‹x ∈ A› this))
```

Notice that it is considerably longer than the informal proof in the last chapter, because we have spelled out every last detail. Unfortunately, this does not necessarily make it more readable. Keep in mind that you can always write long proofs incrementally, using `sorry`. You can also break up long proofs into smaller pieces:

```
theorem inter_union_subset : A ∩ (B ∪ C) ⊆ (A ∩ B) ∪ (A ∩ C) :=
assume x,
assume h : x ∈ A ∩ (B ∪ C),
have x ∈ A, from and.left h,
have x ∈ B ∪ C, from and.right h,
or.elim (‹x ∈ B ∪ C›)
  (assume : x ∈ B,
    have x ∈ A ∩ B, from and.intro ‹x ∈ A› ‹x ∈ B›,
    show x ∈ (A ∩ B) ∪ (A ∩ C), from or.inl this)
  (assume : x ∈ C,
    have x ∈ A ∩ C, from and.intro ‹x ∈ A› ‹x ∈ C›,
    show x ∈ (A ∩ B) ∪ (A ∩ C), from or.inr this)

theorem inter_union_inter_subset : (A ∩ B) ∪ (A ∩ C) ⊆ A ∩ (B ∪ C) :=
assume x,
assume : x ∈ (A ∩ B) ∪ (A ∩ C),
or.elim this
  (assume h : x ∈ A ∩ B,
    have x ∈ A, from and.left h,
    have x ∈ B, from and.right h,
    have x ∈ B ∪ C, from or.inl this,
    show x ∈ A ∩ (B ∪ C), from and.intro ‹x ∈ A› this)
  (assume h : x ∈ A ∩ C,
    have x ∈ A, from and.left h,
    have x ∈ C, from and.right h,
    have x ∈ B ∪ C, from or.inr this,
    show x ∈ A ∩ (B ∪ C), from and.intro ‹x ∈ A› this)

example : A ∩ (B ∪ C) = (A ∩ B) ∪ (A ∩ C) :=
eq_of_subset_of_subset
  (inter_union_subset A B C)
  (inter_union_inter_subset A B C)
```

Notice that the two theorems depend on the variables A, B, and C, which have to be supplied as arguments when they are applied. They also depend on the underlying type, U, but because the variable U was marked implicit, Lean figures it out from the context.

In the last chapter we showed $(A \cap \overline{B}) \cup B = B$. Here is the corresponding proof in Lean:

```
example : (A ∩ -B) ∪ B = A ∪ B :=
calc
  (A ∩ -B) ∪ B = (A ∪ B) ∩ (-B ∪ B) : by rewrite union_distrib_right
           ... = (A ∪ B) ∩ univ     : by rewrite compl_union_self
           ... = A ∪ B              : by rewrite inter_univ
```

Translated to propositions, the theorem above states that for every pair of elements A and B in a Boolean algebra, $(A \wedge \neg B) \vee B = B$. Lean allows us to do calculations on propositions as though they are elements of a Boolean algebra, with equality replaced by $\leftrightarrow$.

```
variables A B : Prop

example : (A ∧ ¬ B) ∨ B ↔ A ∨ B :=
calc
    (A ∧ ¬ B) ∨ B ↔ (A ∨ B) ∧ (¬ B ∨ B) : by rw and_or_distrib_right
              ... ↔ (A ∨ B) ∧ true      : by rw not_or_self
              ... ↔ (A ∨ B)             : by rw and_true
```

12.3 Power Sets and Indexed Families

We can also work with power sets and indexed unions and intersections in Lean. If `A : set U`, then `powerset A` is a subset of `set U`, that is, we have `powerset A : set (set X)`. For Lean, `A ∈ powerset B` means the same thing as `A ⊆ B`, which, in turn, means `∀x, x ∈ A → x ∈ B`.

```
#check powerset A

example : A ∈ powerset (A ∪ B) :=
assume x,
assume : x ∈ A,
show x ∈ A ∪ B, from or.inl ‹x ∈ A›
```

A family of sets in Lean is written as `A : I → set U` where `I` is a `Type`. Then the intersection and union of the family of sets `A` is written `  i, A i  i, A i`. These characters can be typed with `\I` and `\Un`. For Lean, `x ∈  i, A i` means `∀i : I, x ∈ A i` and `x ∈  i, A i` means `∃i : I, x ∈ A i`. To refresh your memory at to how to work with the universal and existential quantifier in Lean, see Chapters 9.

```
variables {I U : Type}
variables (A : I → set U)

example (i₀ : I) : ( i, A i) ⊆ ( i, A i) :=
assume x,
assume h : x ∈  i, A i,
have x ∈ A i₀, from mem_of_mem_Inter h i₀,
show x ∈  i, A i, from mem_Union ‹x ∈ A i₀›
```

12.4 Exercises

1. Fill in the `sorry`'s.

```
import data.set
open set

section
  variable  U : Type
  variable  A : U → Prop
  variable  B : U → U → Prop

  -- problem 1

  example (h : ∀ x y, A x → B x y) : ∀ x, (A x → ∀ y, B x y) :=
  sorry
end

section
  variable U : Type
  variables A B C : set U

  -- problem 2

  example : ∀ x, x ∈ A ∩ C → x ∈ A ∪ B :=
  sorry

  -- problem 3
  example : ∀ x, x ∈ -(A ∪ B) → x ∈ -A :=
  sorry
end
```

2. Fill in the sorry.

```
import logic.basic data.set
open set

variable {U : Type}

/- defining "disjoint" -/

definition disj (A B : set U) : Prop := ∀ ⦃x⦄, x ∈ A → x ∈ B → false

example (A B : set U) (h : ∀ x, ¬ (x ∈ A ∧ x ∈ B)) : disj A B :=
assume x,
assume h1 : x ∈ A,
assume h2 : x ∈ B,
have h3 : x ∈ A ∧ x ∈ B, from and.intro h1 h2,
show false, from h x h3

-- notice that we do not have to mention x when applying h : disj A B
example (A B : set U) (h1 : disj A B) (x : U) (h2 : x ∈ A) (h3 : x ∈ B) : false :=
h1 h2 h3

-- the same is true of ⊆
example (A B : set U) (x : U) (h : A ⊆ B) (h1 : x ∈ A) : x ∈ B :=
h h1

/- problem 1 -/

-- replace the "sorry" by a proof
example (A B C D : set U) (h1 : disj A B) (h2 : C ⊆ A) (h3 : D ⊆ B) : disj C D :=
sorry
```

3. Prove the following facts about indexed unions and intersections.

```
import data.set
open set

variables {I J U : Type}
variables (A : I → J → set U)

example : ( i,  j, A i j) ⊆ ( j,  i, A i j) :=
sorry
```

```
import data.set
open classical set

variables {I U : Type}
variables (A : I → set U) (B : set U)

example : B ∩ ( i, A i) =  i, B ∩ A i :=
sorry

-- Hint: the reverse inclusion of the following example requires classical reasoning
```

```
example : B ∪ ( i, A i) =  i, B ∪ A i :=
sorry
```

4. Prove the following fact about power sets. You can use the theorems `subset.trans` and `subset.refl`

```
import data.set
open set

variables {U : Type}
variables (A B C : set U)

-- For the exercise these two facts are useful
example (h1 : A ⊆ B) (h2 : B ⊆ C) : A ⊆ C :=
subset.trans h1 h2

example : A ⊆ A :=
subset.refl A

example : A ⊆ B ↔ powerset A ⊆ powerset B :=
sorry
```

CHAPTER

THIRTEEN

RELATIONS

In Chapter 7 we discussed the notion of a *relation symbol* in first-order logic, and in Chapter 10 we saw how to interpret such a symbol in a model. In mathematics, we are generally interested in different sorts of relationships between mathematical objects, and so the notion of a relation is ubiquitous. In this chapter, we will consider some common kinds of relations.

In some axiomatic foundations, the notion of a relation is taken to be primitive, but in axiomatic set theory, a relation is taken to be a set of tuples of the corresponding arity. For example, we can take a binary relation on A to be a subset of $A \times A$, where $R(a, b)$ means that $(a, b) \in R$. The foundational definition is generally irrelevant to everyday mathematical practice; what is important is simply that we can write expressions like $R(a, b)$, and that they are true or false, depending on the values of a and b. In mathematics, we often use *infix* notation, writing aRb instead of $R(a, b)$.

13.1 Order Relations

We will start with a class of important binary relations in mathematics, namely, *partial orders*.

Definition. A binary relation $\leq$ on a domain A is a *partial order* if it has the following three properties:

- *reflexivity*: $a \leq a$, for every a in A
- *transitivity*: if $a \leq b$ and $b \leq c$, then $a \leq c$, for every a, b, and c in A
- *antisymmetry*: if $a \leq b$ and $b \leq a$ then $a = b$, for every a and b in A.

Notice the compact way of introducing the symbol $\leq$ in the statement of the definition, and the fact that $\leq$ is written as an infix symbol. Notice also that even though the relation is written with the symbol $\leq$, it is the only symbol occurring in the definition; mathematical practice favors natural language to describe its properties.

You now know enough, however, to recognize the universal quantifiers that are present in the three clauses. In symbolic logic, we would write them as follows:

- $\forall a \; (a \leq a)$
- $\forall a, b, c \; (a \leq b \wedge b \leq c \to a \leq c)$
- $\forall a, b \; (a \leq b \wedge b \leq a \to a = b)$

Here the variables a, b, and c implicitly range over the domain A.

The use of the symbol $\leq$ is meant to be suggestive, and, indeed, the following are all examples of partial orders:

- $\leq$ on the natural numbers
- $\leq$ on the integers
- $\leq$ on the rational numbers
- $\leq$ on the real numbers

But keep in mind that $\leq$ is only a symbol; it can have unexpected interpretations as well. For example, all of the following are also partial orders:

- $\geq$ on the natural numbers
- $\geq$ on the integers
- $\geq$ on the rational numbers
- $\geq$ on the real numbers

These are not fully representative of the class of partial orders, in that they all have an additional property:

Definition. A partial order $\leq$ on a domain A is a *total order* (also called a *linear order*) if it also has the following property:

- for every a and b in A, either $a \leq b$ or $b \leq a$.

You can check these these are two examples of partial orders that are not total orders:

- the divides relation, $x \mid y$, on the integers
- the subset relation, $x \subseteq y$, on sets of elements of some domain A

On the integers, we also have the strict order relation, $<$, which is not a partial order, since it is not reflexive. It is, rather, an instance of a *strict partial order*:

Definition. A binary relation $<$ on a domain A is a *strict partial order* if it satisfies the following:

- *irreflexivity*: $a \not< a$ for every a in A.
- *transitivity*: $a < b$ and $b < c$ implies $a < c$, for every a, b, and c in A.

A strict partial order is a *strict total order* (or *strict linear order*) if, in addition, we have the following property:

- *trichotomy*: $a < b$, $a = b$, or $a > b$ for every a and b in A.

Here, $b \not< a$ means, of course, that it is not the case that $a < b$, and $a > b$ is alternative notation for $b < a$. To distinguish an ordinary partial order from a strict one, an ordinary partial order is sometimes called a *weak* partial order.

Proposition. A strict partial order $<$ on A is *asymmetric*: for every a and b, $a < b$ implies $b \not< a$.

Proof. Suppose $a < b$ and $b < a$. Then, by transitivity, $a < a$, contradicting irreflexivity.

On the integers, there are precise relationships between $<$ and $\leq$: $x \leq y$ if and only if $x < y$ or $x = y$, and $x < y$ if and only if $x \leq y$ and $x \neq y$. This illustrates a more general phenomenon.

Theorem. Suppose $\leq$ is a partial order on a domain A. Define $a < b$ to mean that $a \leq b$ and $a \neq b$. Then $<$ is a strict partial order. Moreover, if $\leq$ is total, so is $<$.

Theorem. Suppose $<$ is a strict partial order on a domain A. Define $a \leq b$ to mean $a < b$ or $a = b$. Then $\leq$ is a partial order. Moreover, if $<$ is total, so is $\leq$.

We will prove the first here, and leave the second as an exercise. This proof is a nice illustration of how universal quantification, equality, and propositional reasoning are combined in a mathematical argument.

Proof. Suppose $\leq$ is a partial order on A, and $<$ be defined as in the statement of the theorem. Irreflexivity is immediate, since $a < a$ implies $a \neq a$, which is a contradiction.

To show transitivity, suppose $a < b$ and $b < c$. Then we have $a \leq b$, $b \leq c$, $a \neq b$, and $b \neq c$. By the transitivity of $\leq$, we have $a \leq c$. To show $a < c$, we only have to show $a \neq c$. So suppose $a = c$. then, from the hypotheses, we have $c < b$ and $b < c$, violating asymmetry. So $a \neq c$, as required.

To establish the last claim in the theorem, suppose $\leq$ is total, and let a and b be any elements of A. We need to show that $a < b$, $a = b$, or $a > b$. If $a = b$, we are done, so we can assume $a \neq b$. Since $\leq$ is total, we have $a \leq b$ or $a \leq b$. Since $a \neq b$, in the first case we have $a < b$, and in the second case, we have $a > b$.

13.2 More on Orderings

Let $\leq$ be a partial order on a domain, A, and let $<$ be the associated strict order, as defined in the last section. It is possible to show that if we go in the other direction, and define $\leq'$ to be the partial order associated to $<$, then $\leq$ and $\leq'$ are the same, which is to say, for every a and b in A, $a \leq b$ if and only if $a \leq' b$. So we can think of every partial order as really being a pair, consisting of a weak partial order and an associated strict one. In other words, we can assume that $x < y$ holds if and only if $x \leq y$ and $x \neq y$, and we can assume $x \leq y$ holds if and only if $x < y$ or $x = y$.

We will henceforth adopt this convention. Given a partial order $\leq$ and the associated strict order $<$, we leave it to you to show that if $x \leq y$ and $y < z$, then $x < z$, and, similarly, if $x < y$ and $y \leq z$, then $x < z$.

Consider the natural numbers with the less-than-or-equal relation. It has a least element, 0. We can express the fact that 0 is the least element in at least two ways:

- 0 is less than or equal to every natural number.
- There is no natural number that is less than 0.

In symbolic logic, we could formalize these statements as follows:

- $\forall x \, (0 \leq x)$
- $\forall x \, (x \not< 0)$

Using the existential quantifier, we could render the second statement more faithfully as follows:

- $\neg \exists x \, (x < 0)$

Notice that this more faithful statement is equivalent to the original, using deMorgan's laws for quantifiers.

Are the two statements above equivalent? Say an element y is *minimum* for a partial order if it is less than or equal to any other element; this is, if it takes the place of 0 in the first statement. Say that an element y is *minimal* for a partial order if no element is less than it; that is, if it takes the place of 0 in the second statement. Two facts are immediate.

Theorem. Any minimum element is minimal.

Proof. Suppose x is minimum for $\leq$. We need to show that x is minimal, that is, for every y, it is not the case that $y < x$. Suppose $y < x$. Since x is minimum, we have $x \leq y$. From $y < x$ and $x \leq y$, we have $y < y$, contradicting the irreflexivity of $<$.

Theorem. If a partial order $\leq$ has a minimum element, it is unique.

Proof. Suppose x_1 and x_2 are both minimum. Then $x_1 \leq x_2$ and $x_2 \leq x_1$. By antisymmetry, $x_1 = x_2$.

Notice that we have interpreted the second theorem as the statement that if x_1 and x_2 are both minimum, then $x_1 = x_2$. Indeed, this is exactly what we mean when we say that something is "unique." When a partial order has a minimum element x, uniqueness is what justifies calling x *the* minimum element. Such an x is also called the *least* element or the *smallest* element, and the terms are generally interchangeable.

The converse to the second theorem – that is, the statement that every minimal element is minimum – is false. As an example, consider the nonempty subsets of the set $\{1, 2\}$ with the subset relation. In other words, consider the collection of sets $\{1\}$, $\{2\}$, and $\{1, 2\}$, where $\{1\} \subseteq \{1, 2\}$, $\{2\} \subseteq \{1, 2\}$, and, of course, every element is a subset of itself. Then you can check that $\{1\}$ and $\{2\}$ are each minimal, but neither is minimum. (One can also exhibit such a partial order by drawing a diagram, with dots labeled a, b, c, etc., and upwards edges between elements to indicate that one is less than or equal to the other.)

Notice that the statement "a minimal element of a partial order is not necessarily minimum" makes an "existential" assertion: it says that there is a partial order $\leq$, and an element x of the domain, such that x is minimal but not minimum. For a fixed partial order $\leq$, we can express the assertion that such an x exists as follows:

$$\exists x\ (\forall y\ (y \not< x) \wedge \forall y\ (x \leq y)).$$

The assertion that there exists a domain A, and a partial order $\leq$ on that domain A, is more dramatic: it is a "higher order" existential assertion. But symbolic logic provides us with the means to make assertions like these as well, as we will see later on.

We can consider other properties of orders. An order is said to be *dense* if between any two distinct elements, there is another element. More precisely, an order is dense if, whenever $x < y$, there is an element z satisfying $x < z$ and $z < y$. For example, the rational numbers are dense with the usual $\leq$ ordering, but not the integers. Saying that an order is dense is another example of an implicit use of existential quantification.

13.3 Equivalence Relations and Equality

In ordinary mathematical language, an *equivalence relation* is defined as follows.

Definition. A binary relation $\equiv$ on some domain A is said to be an *equivalence relation* if it is reflexive, symmetric, and transitive. In other words, $\equiv$ is an equivalent relation if it satisfies these three properties:

- *reflexivity*: $a \equiv a$, for every a in A.

- *symmetry*: if $a \equiv b$, then $b \equiv a$, for every a and b in A.
- *transitivity*: if $a \equiv b$ and $b \equiv c$, then $a \equiv c$, for every a, b, and c in A.

We leave it to you to think about how you could write these statements in first-order logic. (Note the similarity to the rules for a partial order.) We will also leave you with an exercise: by a careful choice of how to instantiate the quantifiers, you can actually prove the three properties above from the following two:

- $\forall a \; (a \equiv a)$
- $\forall a, b, c \; (a \equiv b \wedge c \equiv b \to a \equiv c)$

Try to verify this using natural deduction or Lean.

These three properties alone are not strong enough to characterize equality. You should check that the following informal examples are all instances of equivalence relations:

- the relation on days on the calendar, given by "x and y fall on the same day of the week"
- the relation on people currently alive on the planet, given by "x and y have the same age"
- the relation on people currently alive on the planet, given by "x and y have the same birthday"
- the relation on cities in the United States, given by "x and y are in the same state"

Here are two common mathematical examples:

- the relation on lines in a plane, given by "x and y are parallel"
- for any fixed natural number $m \geq 0$, the relation on natural numbers, given by "x is congruent to y modulo m"

Here, we say that x is congruent to y modulo m if they leave the same remainder when divided by m. Soon, you will be able to prove rigorously that this is equivalent to saying that $x - y$ is divisible by m.

Consider the equivalence relation on citizens of the United States, given by "x and y have the same age." There are some properties that respect that equivalence. For example, suppose I tell you that John and Susan have the same age, and I also tell you that John is old enough to vote. Then you can rightly infer that Susan is old enough to vote. On the other hand, if I tell you nothing more than the facts that John and Susan have the same age and John lives in South Dakota, you cannot infer that Susan lives in South Dakota. This little example illustrates what is special about the *equality* relation: if two things are equal, then they have exactly the same properties.

An important related notion is that of an *equivalence class*. Let $\equiv$ be an equivalence relation on a set A. For every element a in A, let $[a]$ be the set of elements $\{c \mid c \equiv a\}$, that is, the set of elements of A that are equivalent to a. We call $[a]$ the equivalence class of A, under the equivalence relation $\equiv$.

Equivalence tries to capture a "weak" notion of equality: if two elements of A are equivalent, they are not necessarily the same, but they are "similar" in some way. Equivalence classes collect similar objects together. If we define $A' = \{[a] : a \in A\}$, the set of equivalence classes of elements in A, we get a version of the set A where sets of similar elements have been "compressed" into single elements. This is illustrated in an exercise below.

13.4 Exercises

1. Suppose $<$ is a strict partial order on a domain A, and define $a \leq b$ to mean that $a < b$ or $a = b$.
   - Show that $\leq$ is a partial order.
   - Show that if $<$ is moreover a strict total order, then $\leq$ is a total order.

(Above we proved the analogous theorem going in the other direction.)

2. Suppose $<$ is a strict partial order on a domain A. (In other words, it is transitive and asymmetric.) Suppose that $\leq$ is defined so that $a \leq b$ if and only if $a < b$ or $a = b$. We saw in class that $\leq$ is a partial order on a domain A, i.e.~it is reflexive, transitive, and antisymmetric.

   Prove that for every a and b in A, we have $a < b$ iff $a \leq b$ and $a \neq b$, using the facts above.

3. An *ordered graph* is a collection of vertices (points), along with a collection of arrows between vertices. For each pair of vertices, there is at most one arrow between them: in other words, every pair of vertices is either unconnected, or one vertex is "directed" toward the other. Note that it is possible to have an arrow from a vertex to itself.

   Define a relation $\leq$ on the set of vertices, such that for two vertices a and b, $a \leq b$ means that there is an arrow from a pointing to b.

   On an arbitrary graph, is $\leq$ a partial order, a strict partial order, a total order, a strict total order, or none of the above? If possible, give examples of graphs where $\leq$ fails to have these properties.

4. Let $\equiv$ be an equivalence relation on a set A. For every element a in A, let $[a]$ be the equivalence class of a: that is, the set of elements $\{c \mid c \equiv a\}$. Show that for every a and b, $[a] = [b]$ if and only if $a \equiv b$.

   (Hints and notes:

   - Remember that since you are proving an ‘‘if and only if’’ statement, there are two directions to prove.
   - Since that $[a]$ and $[b]$ are sets, $[a] = [b]$ means that for every element c, c is in $[a]$ if and only if c is in $[b]$.
   - By definition, an element c is in $[a]$ if and only if $c \equiv a$. In particular, a is in $[a]$.)

5. Let the relation $\sim$ on the natural numbers $\mathbb{B}$ be defined as follows: if n is even, then $n \sim n + 1$, and if n is odd, then $n \sim n - 1$. Furthermore, for every n, $n \sim n$. Show that $\sim$ is an equivalence relation. What is the equivalence class of the number 5? Describe the set of equivalence classes $\{[n] \mid n \in \mathbb{N}\}$.

6. Show that the relation on lines in the plane, given by "l_1 and l_2 are parallel," is an equivalence relation. What is the equivalence class of the x-axis? Describe the set of equivalence classes $\{[l] \mid l \text{ is a line in the plane}\}$.

7. A binary relation $\leq$ on a domain A is said to be a *preorder* it is is reflexive and transitive. This is weaker than saying it is a partial order; we have removed the requirement that the relation is asymmetric. An example is the ordering on people currently alive on the planet defined by setting $x \leq y$ if and only if x ‘s birth date is earlier than y ‘s. Asymmetry fails, because different people can be born on the same day. But, prove that the following theorem holds:

   Theorem. Let $\leq$ be a preorder on a domain A. Define the relation $\equiv$, where $x \equiv y$ holds if and only if $x \leq y$ and $y \leq x$. Then $\equiv$ is an equivalence relation on A.

CHAPTER

FOURTEEN

RELATIONS IN LEAN

In the last chapter, we noted that set theorists think of a binary relation R on a set A as a set of ordered pairs, so that $R(a, b)$ really means $(a, b) \in R$. An alternative is to think of R as a function which, when applied to a and B, returns the proposition that $R(a, b)$ holds. This is the viewpoint adopted by Lean: a binary relation on a type `A` is a function `A → A → Prop`. So, if `R` is a binary relation on `A` and we have `a b : A`, then `R a b` is a proposition.

As in informal mathematics, we often wish to use infix notation for relations. We will see below that Lean supports this practice.

14.1 Order Relations

We can reason about partial orders in Lean by fixing a type, `A`, and a binary relation, `R`, and working under the hypotheses that `A` is reflexive, transitive, and antisymmetric:

```
section
  parameters {A : Type} {R : A → A → Prop}
  parameter (reflR : reflexive R)
  parameter (transR : transitive R)
  parameter (antisymmR : ∀ {a b : A}, R a b → R b a → a = b)

  local infix ≤ := R
end
```

The `parameter` and `parameter` commands are similar to the `variable` and `premise` commands, except that parameters are fixed within a section. In other words, if you prove a theorem about `R` in the section above, you cannot apply that theorem to another relation, `S`, without closing the section. Since the parameter `R` is fixed, Lean allows us to define notation for `R`, to be used locally in the section.

In the example below, having fixed a partial order, `R`, we define the corresponding strict partial order and prove that it is, indeed, a strict order.

```
section
parameters {A : Type} {R : A → A → Prop}
parameter (reflR : reflexive R)
parameter (transR : transitive R)
parameter (antisymmR : ∀ {a b : A}, R a b → R b a → a = b)

local infix ≤ := R

definition R' (a b : A) : Prop := a ≤ b ∧ a ≠ b

local infix < := R'
```

```
theorem irrefl (a : A) : ¬ a < a :=
assume : a < a,
have a ≠ a, from and.right this,
have a = a, from rfl,
show false, from ‹a ≠ a› ‹a = a›

theorem trans {a b c : A} (h₁ : a < b) (h₂ : b < c) : a < c :=
have a ≤ b, from and.left h₁,
have a ≠ b, from and.right h₁,
have b ≤ c, from and.left h₂,
have b ≠ c, from and.right h₂,
have a ≤ c, from transR ‹a ≤ b› ‹b ≤ c›,
have a ≠ c, from
    assume : a = c,
    have c ≤ b, from ‹a = c› ▸ ‹a ≤ b›,
    have b = c, from antisymmR ‹b ≤ c› ‹c ≤ b›,
    show false, from ‹b ≠ c› ‹b = c›,
show a < c, from and.intro ‹a ≤ c› ‹a ≠ c›
end
```

Notice that we have used the command `open eq.ops` to avail ourselves of the extra notation for equality proofs, so that the expression `` `a = c` ▸ `a ≤ b` `` denotes a proof of `c ≤ b`.

In the exercises, we ask you to show the other direction of this: from a strict partial order we can define a partial order.

14.2 Orderings on Numbers

Conveniently, Lean has the normal orderings on the natural numbers, integers, and so on defined already.

```
open nat
variables n m : ℕ

#check 0 ≤ n
#check n < n + 1

example : 0 ≤ n := zero_le n
example : n < n + 1 := lt_succ_self n

example (h : n + 1 ≤ m) : n < m + 1 :=
have h1 : n < n + 1, from lt_succ_self n,
have h2 : n < m, from lt_of_lt_of_le h1 h,
have h3 : m < m + 1, from lt_succ_self m,
show n < m + 1, from lt.trans h2 h3
```

There are many theorems in Lean that are useful for proving facts about inequality relations. We list some common ones here.

```
variables (A : Type) [partial_order A]
variables a b c : A

#check (le_trans : a ≤ b → b ≤ c → a ≤ c)
#check (lt_trans : a < b → b < c → a < c)
#check (lt_of_lt_of_le : a < b → b ≤ c → a < c)
```

```
#check (lt_of_le_of_lt : a ≤ b → b < c → a < c)
#check (le_of_lt : a < b → a ≤ b)
```

Here the declaration at the top says that `A` has the structure of a partial order. There are also properties that are specific to some domains, like the natural numbers:

```
variable n : ℕ

#check (nat.zero_le : ∀ n : ℕ, 0 ≤ n)
#check (nat.lt_succ_self : ∀ n : ℕ, n < n + 1)
#check (nat.le_succ : ∀ n : ℕ, n ≤ n + 1)
```

14.3 Exercises

1. Replace the `sorry` commands in the following proofs to show that we can create a partial order `R'` out of a strict partial order `R`.

```
section
parameters {A : Type} {R : A → A → Prop}
parameter (irreflR : irreflexive R)
parameter (transR : transitive R)

local infix < := R

definition R' (a b : A) : Prop := R a b ∨ a = b
local infix ≤ := R'

theorem reflR' (a : A) : a ≤ a := sorry
theorem transR' {a b c : A} (h1 : a ≤ b) (h2 : b ≤ c): a ≤ c := sorry
theorem antisymmR' {a b : A} (h1 : a ≤ b) (h2 : b ≤ a) : a = b := sorry
end
```

2. Complete the following proof. Note: we write `(1 : ℕ)` instead of just `1` so that Lean does not confuse the natural number `1` with the integer, rational, or so on.

```
open nat

example : (1 : ℕ) ≤ (4 : ℕ) :=
sorry
```

3. Only one of the following two theorems is provable. Figure out which one is true, and replace the `sorry` command with a complete proof.

```
section
  parameters {A : Type} {a b c : A} {R : A → A → Prop}
  parameter (Rab : R a b)
  parameter (Rbc : R b c)
  parameter (nRac : ¬ R a c)

  -- Prove one of the following two theorems:

  theorem R_is_strict_partial_order : irreflexive R ∧ transitive R :=
  sorry

  theorem R_is_not_strict_partial_order : ¬(irreflexive R ∧ transitive R) :=
  sorry
end
```

CHAPTER

FIFTEEN

FUNCTIONS

In the late nineteenth century, developments in a number of branches of mathematics pushed towards a uniform treatment of sets, functions, and relations. We have already considered sets and relations. In this chapter, we consider functions and their properties.

A function, f, is ordinary understood as a mapping from a domain X to another domain Y. In set-theoretic foundations, X and Y are arbitrary sets. We have seen that in a type-based system like Lean, it is natural to distinguish between types and subsets of a type. In other words, we can consider a type `X` of elements, and a set `A` of elements of that type. Thus, in the type-theoretic formulation, it is natural to consider functions between types `X` and `Y`, and consider their behavior with respect to subsets of `X` and `Y`.

In everyday mathematics, however, set-theoretic language is common, and most mathematicians think of a function as a map between sets. When discussing functions from a mathematical standpoint, therefore, we will also adopt this language, and later switch to the type-theoretic representation when we talk about formalization in Lean.

15.1 The Function Concept

If X and Y are any sets, we write $f : X \to Y$ to express the fact that f is a function from X to Y. This means that f assigns a value $f(x)$ in Y to every element x of X. The set X is called the *domain* of f, and the set Y is called the *codomain*. (Some authors use the word "range" for the codomain, but today it is more common to use the word "range" for what we call the *image* of A below. We will avoid the ambiguity by avoiding the word range altogether.)

The simplest way to define a function is to give its value at every x with an explicit expression. For example, we can write any of the following:

- Let $f : \mathbb{N} \to \mathbb{N}$ be the function defined by $f(n) = n + 1$.
- Let $g : \mathbb{R} \to \mathbb{R}$ be the function defined by $g(x) = x^2$.
- Let $h : \mathbb{N} \to \mathbb{N}$ be the function defined by $h(n) = n^2$.
- Let $k : \mathbb{N} \to \{0, 1\}$ be the function defined by

$$k(n) = \begin{cases} 0 & \text{if } n \text{ is even} \\ 1 & \text{if } n \text{ is odd.} \end{cases}$$

The ability to define functions using an explicit expression raises the foundational question as to what counts as legitimate "expression." For the moment, let us set that question aside, and simply note that modern mathematics is comfortable with all kinds of exotic definitions. For example, we can define a function $f : \mathbb{R} \to \{0, 1\}$ by

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

This is at odds with a view of functions as objects that are computable in some sense. It is not at all clear what it means to be presented with a real number as input, let alone whether it is possible to determine, algorithmically, whether such a number is rational or not. We will return to such issues in a later chapter.

Notice that the choice of the variables x and n in the definitions above are arbitrary. They are bound variables in that the functions being defined do not depend on x or n. The values remain the same under renaming, just as the truth values of "for every x, $P(x)$" and "for every y, $P(y)$" are the same. Given an expression $e(x)$ that depends on the variable x, logicians often use the notation $\lambda x\ e(x)$ to denote the function that maps x to $e(x)$. This is called "lambda notation," for the obvious reason, and it is often quite handy. Instead of saying "let f be the function defined by $f(x) = x + 1$," we can say "let $f = \lambda x\ (x + 1)$." This is *not* common mathematical notation, and it is best to avoid it unless you are talking to logicians or computer scientists. We will see, however, that lambda notation is built in to Lean.

For any set X, we can define a function $i_X(x)$ by the equation $i_X(x) = x$. This function is called the *identity function*. More interestingly, let $f : X \to Y$ and $g : Y \to Z$. We can define a new function $k : X \to Z$ by $k(x) = g(f(x))$. The function k is called *the composition of :math:'f' and g* or *:math:'f' composed with g* and it is written $g \circ f$. The order is somewhat confusing; you just have to keep in mind that to evaluate the expression $g(f(x))$ you first evaluate f on input x, and then evaluate g.

We think of two functions $f, g : X \to Y$ as being equal, or the same function, when for they have the same values on every input; in other words, for every x in X, $f(x) = g(x)$. For example, if $f, g : \mathbb{R} \to \mathbb{R}$ are defined by $f(x) = x + 1$ and $g(x) = 1 + x$, then $f = g$. Notice that the statement that two functions are equal is a universal statement (that is, for the form "for every x, …").

Proposition. For every $f : X \to Y$, $f \circ i_X = f$ and $i_Y \circ f = f$.

Proof. Let x be any element of X. Then $(f \circ i_X)(x) = f(i_X(x)) = f(x)$, and $(i_Y \circ f)(x) = i_Y(f(x)) = x$.

Suppose $f : X \to Y$ and $g : Y \to X$ satisfy $g \circ f = i_X$. Remember that this means that $g(f(x)) = x$ for every x in X. In that case, g is said to be a *left inverse* to f, and f is said to be a *right inverse* to g. Here are some examples:

- Define $f, g : \mathbb{R} \to \mathbb{R}$ by $f(x) = x + 1$ and $g(x) = x - 1$. Then g is both a left and a right inverse to f, and vice-versa.
- Write $\mathbb{R}^{\geq 0}$ to denote the nonnegative reals. Define $f : \mathbb{R} \to \mathbb{R}^{\geq 0}$ by $f(x) = x^2$, and define $g : \mathbb{R}^{\geq 0} \to \mathbb{R}$ by $g(x) = \sqrt{x}$. Then $f(g(x)) = (\sqrt{x})^2 = x$ for every x in the domain of g, so f is a left inverse to g, and g is a right inverse to f. On the other hand, $g(f(x)) = \sqrt{x^2} = |x|$, which is not the same as x when x is negative. So g is not a left inverse to f, and f is not a right inverse to g.

The following fact is not at all obvious, even though the proof is short:

Proposition. Suppose $f : X \to Y$ has a left inverse, h, and a right inverse k. Then $h = k$.

Proof. Let y be any element in Y. The idea is to compute $h(f(k(y))$ in two different ways. Since h is a left inverse to f, we have $h(f(k(y))) = k(y)$. On the other hand, since k is a right inverse to f, $f(k(y)) = y$, and so $h(f(k(y)) = h(y)$. So $k(y) = h(y)$.

If g is both a right and left inverse to f, we say that g is simply the inverse of f. A function f may have more than one left or right inverse (we leave it to you to cook up examples), but it can have at most one inverse.

Proposition. Suppose $g_1, g_2 : Y \to X$ are both inverses to f. Then $g_1 = g_2$.

Proof. This follows from the previous proposition, since (say) g_1 is a left inverse to f, and g_2 is a right inverse.

When f has an inverse, g, this justifies calling g *the* inverse to f, and writing f^{-1} to denote g. Notice that if f^{-1} is an inverse to f, then f is an inverse to f^{-1}. So if f has an inverse, then so does f^{-1}, and $(f^{-1})^{-1} = f$. For any set A, clearly we have $i_X^{-1} = i_X$.

Proposition. Suppose $f : X \to Y$ and $g : Y \to Z$. If $h : Y \to X$ is a left inverse to f and $k : Z \to Y$ is a left inverse to g, then $h \circ k$ is a left inverse to $g \circ f$.

Proof. For every x in X,

$$(h \circ k) \circ (g \circ f)(x) = h(k(g(f(x)))) = h(f(x)) = x.$$

Corollary. The previous proposition holds with "left" replaced by "right."

Proof. Switch the role of f with h and g with k in the previous proposition.

Corollary. If $f : X \to Y$ and $g : Y \to Z$ both have inverses, then $(f \circ g)^{-1} = g^{-1} \circ f^{-1}$.

15.2 Injective, Surjective, and Bijective Functions

A function $f : X \to Y$ is said to be *injective*, or an *injection*, or *one-one*, if given any x_1 and x_2 in A, if $f(x_1) = f(x_2)$, then $x_1 = x_2$. Notice that the conclusion is equivalent to its contrapositive: if $x_1 \neq x_2$, then $f(x_1) \neq f(x_2)$. So f is injective if it maps distinct element of X to distinct elements of Y.

A function $f : X \to Y$ is said to be *surjective*, or a *surjection*, or *onto*, if for every element y of Y, there is an x in X such that $f(x) = y$. In other words, f is surjective if every element in the codomain is the value of f at some element in the domain.

A function $f : X \to Y$ is said to be *bijective*, or a *bijection*, or a *one-to-one correspondence*, if it is both injective and surjective. Intuitively, if there is a bijection between X and Y, then X and Y have the same size, since f makes each element of X correspond to exactly one element of Y and vice-versa. For example, it makes sense to interpret the statement that there were four Beatles as the statement that there is a bijection between the set $\{1, 2, 3, 4\}$ and the set {John, Paul, George, Ringo}. If we claimed that there were *five* Beatles, as evidenced by the function f which assigns 1 to John, 2 to Paul, 3 to George, 4 to Ringo, and 5 to John, you should object that we double-counted John — that is, f is not injective. If we claimed there were only three Beatles, as evidenced by the function f which assigns 1 to John, 2 to Paul, and 3 to George, you should object that we left out poor Ringo — that is, f is not surjective.

The next two propositions show that these notions can be cast in terms of the existence of inverses.

Proposition. Let $f : X \to Y$.

- If f has a left inverse, then f is injective.
- If f has a right inverse, then f is surjective.
- If f has an inverse, then it is f bijective.

Proof. For the first claim, suppose f has a left inverse g, and suppose $f(x_1) = f(x_2)$. Then $g(f(x_1)) = g(f(x_2))$, and so $x_1 = x_2$.

For the second claim, suppose f has a right inverse h. Let y be any element of Y, and let $x = g(y)$. Then $f(x) = f(g(y)) = y$.

The third claim follows from the first two.

The following proposition is more interesting, because it requires us to define new functions, given hypotheses on f.

Proposition. Let $f : X \to Y$.

- If X is nonempty and f is injective, then f has a left inverse.
- If f is surjective, then f has a right inverse.
- If f if bijective, then it has an inverse.

Proof. For the first claim, let $\hat{x}$ be any element of X, and suppose f is injective. Define $g : Y \to X$ by setting $g(y)$ equal to any x such that $f(x) = y$, if there is one, and $\hat{x}$ otherwise. Now, suppose $g(f(x)) = x'$. By the definition of g, x' has to have the property that $f(x) = f(x')$. Since f is injective, $x = x'$, so $g(f(x)) = x$.

For the second claim, because f is surjective, we know that for every y in Y there is any x such that $f(x) = y$. Define $h : B \to A$ by again setting $h(y)$ equal to any such x. (In contrast to the previous paragraph, here we know that such an x exists, but it might not be unique.) Then, by the definition of h, we have $f(h(y)) = y$.

Notice that the definition of g in the first part of the proof requires the function to "decide" whether there is an x in X such that $f(x) = y$. There is nothing mathematically dubious about this definition, but if many situations, this cannot be done *algorithmically*; in other words, g might not be computable from the data. More interestingly, the definition of h in the second part of the proof requires the function to "choose" a suitable value of x from among potentially many candidates. We will see later that this is a version of the *axiom of choice*. In the early twentieth century, the use of the axiom of choice in mathematics was hotly debated, but today it is commonplace.

Using these equivalences and the results in the previous section, we can prove the following:

Proposition. Let $f : X \to B$ and $g : Y \to Z$.

- If f and g are injective, then so is $g \circ f$.
- If f and g are surjective, then so is $g \circ f$.

Proof. If f and g are injective, then they have left inverses h and k, respectively, in which case $h \circ k$ is a left inverse to $g \circ f$. The second statement is proved similarly.

We can prove these two statements, however, without mentioning inverses at all. We leave that to you as an exercise.

Notice that the expression $f(n) = 2n$ can be used to define infinitely many functions with domain $\mathbb{N}$, such as:

- a function $f : \mathbb{N} \to \mathbb{N}$
- a function $f : \mathbb{N} \to \mathbb{R}$

- a function $f : \mathbb{N} \to \{n \mid n \text{ is even}\}$

Only the third one is surjective. Thus a specification of the function's codomain as well as the domain is essential to making sense of whether a function is surjective.

15.3 Functions and Subsets of the Domain

Suppose f is a function from X to Y. We may wish to reason about the behavior of f on some subset A of X. For example, we can say that f *is injective on* A if for every x_1 and x_2 in A, if $f(x_1) = f(x_2)$, then $x_1 = x_2$.

If f is a function from X to Y and A is a subset of X, we write $f[A]$ to denote the *image of* f *on* A, defined by

$$f[A] = \{y \in Y \mid y = f(x) \text{ for some } x \text{ in } A\}.$$

In words, $f[A]$ is the set of elements of Y that are "hit" by elements of A under the mapping f. Notice that there is an implicit existential quantifier here, so that reasoning about images invariables involves the corresponding rules.

Proposition. Suppose $f : X \to Y$, and A is a subset of X. Then for any x in A, $f(x)$ is in $f[A]$.

Proof. By definition, $f(x)$ is in $f[A]$ if and only if there is some x' in A such that $f(x') = f(x)$. But that holds for $x' = x$.

Proposition. Suppose $f : X \to Y$ and $g : Y \to Z$. Let A be a subset of X. Then

$$(g \circ f)[A] = g[f[A]].$$

Proof. Suppose z is in $(g \circ f)[A]$. Then for some $x \in A$, $z = (g \circ f)(x) = g(f(x))$. By the previous proposition, $f(x)$ is in $f[A]$. Again by the previous proposition, $g(f(x))$ is in $g[f[A]]$.

Conversely, suppose z is in $g[f[A]]$. Then there is a y in $f[A]$ such that $f(y) = z$, and since y is in $f[D]$, there is an x in A such that $f(x) = y$. But then $(g \circ f)(x) = g(f(x)) = g(y) = z$, so z is in $(g \circ f)[A]$.

Notice that if f is a function from X to Y, then f is surjective if and only if $f[X] = Y$. So the previous proposition is a generalization of the fact that the composition of surjective functions is surjective.

Suppose f is a function from X to Y, and A is a subset of X. We can *view* f as a function from A to Y, by simply ignoring the behavior of f on elements outside of A. Properly speaking, this is another function, denoted $f \upharpoonright$ and called "the restriction of f to A." In other words, given $f : X \to Y$ and $A \subseteq X$, $f \upharpoonright A : A \to Y$ is the function defined by $(f \upharpoonright A)(x) = x$ for every x in A. Notice that now "f is injective on A" means simply that the restriction of f to A is injective.

There is another important operation on functions, known as the *preimage*. If $f : X \to Y$ and $B \subseteq Y$, then the *preimage of :math:'B' under f*, denoted $f^{-1}[B]$, is defined by

$$f^{-1}[B] = \{x \in X \mid f(x) \in B\},$$

that is, the set of elements of X that get mapped into B. Notice that this makes sense even if f does not have an inverse; for a given y in B, there may be no x's with the property $f(x) \in B$, or there may be many. If f has an inverse, f^{-1}, then for every y in B there is exactly one $x \in X$ with the property $f(x) \in B$, in which case, $f^{-1}[B]$ means the same thing whether you interpret it as the image of B under f^{-1} or the preimage of B under f.

Proposition. Suppose $f : X \to Y$ and $g : Y \to Z$. Let C be a subset of Z. Then

$$(g \circ f)^{-1}[C] = g^{-1}[f^{-1}[A]].$$

Here we give a long list of facts properties of images and preimages. Here, f denotes an arbitrary function from X to Y, $A, A_1, A_2, \ldots$ denote arbitrary subsets of X, and $B, B_1, B_2, \ldots$ denote arbitrary subsets of Y.

- $A \subseteq f^{-1}[f[A]]$, and if f is injective, $A = f^{-1}[f[A]]$.
- $f[f^{-1}[B]] \subseteq B$, and if f is surjective, $B = f[f^{-1}[B]]$.
- If $A_1 \subseteq A_2$, then $f[A_1] \subseteq f[A_2]$.
- If $B_1 \subseteq B_2$, then $f^{-1}[B_1] \subseteq f^{-1}[B_2]$.
- $f[A_1 \cup A_2] = f[A_1] \cup f[A_2]$.
- $f^{-1}[B_1 \cup B_2] = f^{-1}[B_1] \cup f^{-1}[B_2]$.
- $f[A_1 \cap A_2] \subseteq f[A_1] \cap f[A_2]$, and if f is injective, $f[A_1 \cap A_2] = f[A_1] \cap f[A_2]$.
- $f^{-1}[B_1 \cap B_2] = f^{-1}[B_1] \cap f^{-1}[B_2]$.
- $f[A] \setminus f[B] \subseteq f[A \setminus B]$.
- $f^{-1}[A] \setminus f^{-1}[B] \subseteq f[A \setminus B]$.
- $f[A] \cap B = f[A \cap f^{-1}[B]]$.
- $f[A] \cup B \supseteq f[A \cup f^{-1}[B]]$.
- $A \cap f^{-1}[B] \subseteq f^{-1}[f[A] \cap B]$.
- $A \cup f^{-1}[B] \subseteq f^{-1}[f[A] \cup B]$.

Proving identities like this is typically a matter of unfolding definitions and using basic logical inferences. Here is an example.

Proposition. Let X and Y be sets, $f : X \to Y$, $A \subseteq X$, and $B \subseteq Y$. Then $f[A] \cap B = f[A \cap f^{-1}[B]]$.

Proof. Suppose $y \in f[A] \cap B$. Then $y \in B$, and for some $x \in A$, $f(x) = y$. But this means that x is in $f^{-1}[B]$, and so $x \in A \cap f^{-1}[B]$. Since $f(x) = y$, we have $y \in f[A \cap f^{-1}[B]]$, as needed.

Conversely, suppose $y \in f[A \cap f^{-1}[B]]$. Then for some $x \in A \cap f^{-1}[B]$, we have $f(x) = y$. For this x, have $x \in A$ and $f(x) \in B$. Since $f(x) = y$, we have $y \in B$, and since $x \in A$, we also have $y \in f[A]$, as required.

15.4 Functions and Relations

A binary relation $R(x, y)$ on A and B is *functional* if for every x in A there exists a unique y in B such that $R(x, y)$. If R is a functional relation, we can define a function $f_R : X \to B$ by setting $f_R(x)$ to be equal to the unique y in B such that $R(x, y)$. Conversely, it is not hard to see that if $f : X \to B$ is any function, the relation $R_f(x, y)$ defined by $f(x) = y$ is a functional relation. The relation $R_f(x, y)$ is known as the *graph* of f.

It is not hard to check that functions and relations travel in pairs: if f is the function associated with a functional relation R, then R is the functional relation associated the function f, and vice-versa. In set-theoretic foundations, a function is often defined *to be* a functional relation. Conversely, we have seen that in type-theoretic foundations like the one adopted by Lean, relations are often defined to be certain types of functions. We will discuss these matters later on, and in the meanwhile only remark that in everyday mathematical practice, the foundational details are not so important; what is important is simply that every function has a graph, and that any functional relation can be used to define a corresponding function.

So far, we have been focusing on functions that take a single argument. We can also consider functions $f(x, y)$ or $g(x, y, z)$ that take multiple arguments. For example, the addition function $f(x, y) = x + y$ on the integers takes two integers and returns an integer. Remember, we can consider binary functions, ternary functions, and so on, and the number of arguments to a function is called its "arity." One easy way to make sense of functions with multiple arguments is to think of them as unary functions from a cartesian product. We can think of a function f which takes two arguments, one in A and one in B, and returns an argument in C as a unary function from $A \times B$ to C, whereby $f(a, b)$ abbreviates $f((a, b))$. We have seen that in dependent type theory (and in Lean) it is more convenient to think of such a function f as a function which takes an element of A and returns a function from $B \to C$, so that $f(a, b)$ abbreviates $(f(a))(b)$. Such a function f maps A to $B \to C$, where $B \to C$ is the set of functions from B to C.

We will return to these different ways of modeling functions of higher arity later on, when we consider set-theoretic and type-theoretic foundations. One again, we remark that in ordinary mathematics, the foundational details to not matter much. The two choices above are inter-translatable, and sanction the same principles for reasoning about functions informally.

In mathematics, we often also consider the notion of a *partial function* from X to Y, which is really a function from some subset of X to Y. The fact that f is a partial function from X to Y is sometimes written $f : X \nrightarrow Y$, which should be interpreted as saying that $f : A \to Y$ for some subset A of Y. Intuitively, we think of f as a function from $X \to Y$ which is simply "undefined" at some of its inputs; for example, we can think of $f : \mathbb{R} \nrightarrow \mathbb{R}$ defined by $f(x) = 1/x$, which is undefined at $x = 0$, so that in reality $f : \mathbb{R} \setminus \{0\} \to R$. The set A is sometimes called the *domain of f*, in which case, there is no good name for X; others continue to call X the domain, and refer to A as the *domain of definition*. To indicate that a function f is defined at x, that is, that x is in the domain of definition of f, we sometimes write $f(x) \downarrow$. If f and g are two partial functions from X to Y, we write $f(x) \simeq g(x)$ to mean that either f and g are both defined at x and have the same value, or are both undefined at x. Notions of injectivity, surjectivity, and composition are extended to partial functions, generally as you would expect them to be.

In terms of relations, a partial function f corresponds to a relation $R_f(x, y)$ such that for every x there is at most one y such that $R_f(x, y)$ holds. Mathematicians also sometimes consider *multifunctions* from X to Y, which correspond to relations $R_f(x, y)$ such that for every x in X, there is *at least* one y such that $R_f(x, y)$ holds. There may be many such y; you can think of these as functions which have more than one input value. If you think about it for a moment, you will see that a *partial multifunction* is essentially nothing more than an arbitrary relation.

15.5 Exercises

1. Let f be any function from X to Y, and let g be any function from Y to Z.
   - Show that if $g \circ f$ is injective, then f is injective.
   - Give an example of functions f and g as above, such that that $g \circ f$ is injective, but g is not injective.
   - Show that if $g \circ f$ is injective and f is surjective, then g is injective.
2. Let f and g be as in the last problem. Suppose $g \circ f$ is surjective.
   - Is f necessarily surjective? Either prove that it is, or give a counterexample.

   - Is g necessarily surjective? Either prove that it is, or give a counterexample.

3. A function f from $\mathbb{R}$ to $\mathbb{R}$ is said to be *strictly increasing* if whenever $x_1 < x_2$, $f(x_1) < f(x_2)$.
   - Show that if $f : \mathbb{R} \to \mathbb{R}$ is strictly increasing, then it is injective (and hence it has a left inverse).
   - Show that if $f : \mathbb{R} \to \mathbb{R}$ is strictly increasing, and g is a right inverse to f, then g is strictly increasing.
4. Let $f : X \to Y$ be any function, and let A and B be subsets of X. Show that $f[A \cup B] = f[A] \cup f[B]$.
5. Let $f : X \to Y$ be any function, and let A and B be any subsets of X. Show $f[A] \setminus f[B] \subseteq f[A \setminus B]$.
6. Define notions of composition and inverse for binary relations that generalize the notions for functions.

CHAPTER

SIXTEEN

FUNCTIONS IN LEAN

16.1 Functions and Symbolic Logic

Let us now consider functions in formal terms. Even though we have avoided the use of quantifiers and logical symbols in the definitions in the last chapter, by now you should be seeing them lurking beneath the surface. That fact that two functions $f, g : X \to Y$ are equal if and only if they take the same values at every input can be expressed as follows:

$$\forall x \in X \; (f(x) = g(x)) \leftrightarrow f = g$$

This principle is a known as *function extensionality*, analogous to the principle of extensionality for sets, discussed in Section 12.1. Recall that the notation $\forall x \in X \; P(x)$ abbreviates $\forall x \; (x \in X \to P(x))$, and $\exists x \in X \; P(x)$ abbreviates $\exists x \; (x \in X \wedge P(x))$, thereby relativizing the quantifiers to A.

We can avoid set-theoretic notation if we assume we are working in a logical formalism with basic types for X and Y, so that we can specify that x ranges over X. In that case, we will write instead

$$\forall x : X \; (f(x) = g(x) \leftrightarrow f = g)$$

to indicate that the quantification is over X. Henceforth, we will assume that all our variables range over some type, though we will sometimes omit the types in the quantifiers when they can be inferred from context.

The function f is injective if it satisfies

$$\forall x_1, x_2 : X \; (f(x_1) = f(x_2) \to x_1 = x_2),$$
$$and : math : `f`issurjectiveif$$

$$\forall y : Y \; \exists x : X \; f(x) = y.$$

If $f : X \to Y$ and $g : Y \to X$, g is a left inverse to f if

$$\forall x : X \; g(f(x)) = a.$$

Notice that this is a universal statement, and it is equivalent to the statement that f is a right inverse to g.

Remember that in logic it is common to use lambda notation to define functions. We can denote the identity function by $\lambda x \; x$, or perhaps $\lambda x : X \; x$ to emphasize that the domain of the function is X. If $f : X \to Y$ and $g : Y \to Z$, we can define the composition $g \circ f$ by $g \circ f = \lambda x : X \; g(f(x))$.

Also remember that if $P(x)$ is any predicate, then in first order logic we can assert that there exists a unique x satisfying $P(x)$, written $\exists! x \; P(x)$, with the conjunction of the following two statements:

- $\exists x \; P(x)$
- $\forall x_1, x_2 \; (P(x_1) \wedge P(x_2) \to x_1 = y_1)$

Equivalently, we can write

$$\exists (P(x) \wedge \forall x' \; (P(x') \to x' = x)).$$

Assuming $\exists ! x \; P(x)$, the following two statements are equivalent:

- $\exists x \; (P(x) \wedge Q(x))$
- $\forall x \; (P(x) \to Q(x))$

and both can be taken to assert that "the x satisfying P also satisfies Q."

A binary relation R on X and Y is functional if it satisfies

$$\forall x \; \exists ! y \; R(x, y).$$

In that case, a logician might use "iota notation,"

$$f(x) = \iota y \; R(x, y)$$

to define $f(x)$ to be equal to the unique y satisfying $R(x, y)$. If R satisfies the weaker property

$$\forall x \; \exists y \; R(x, y),$$

a logician might use "the Hilbert epsilon" to define a function

$$f(x) = \varepsilon y \; R(x, y)$$

to "choose" a value of y satisfying $R(x, y)$. As we have noted above, this is an implicit use of the axiom of choice.

16.2 Second- and Higher-Order Logic

In contrast to first-order logic, where we start with a fixed stock of function and relation symbols, the topics we have been considering in the last few chapters encourage us to consider a more expressive language with variables ranging over functions and relations as well. For example, saying that a function $f : X \to Y$ has a left-inverse implicitly involves a quantifying over functions,

$$\exists g \; \forall x \; g(f(x)) = x.$$

The theorem that asserts that if any function f from X to Y is injective then it has a left-inverse can be expressed as follows:

$$\forall x_1, x_2 \; (f(x_1) = f(x_2) \to x_1 = x_2) \to \exists g \; \forall x \; g(f(x)) = x.$$

Similarly, saying that two sets X and Y have a one-to-one correspondence asserts the existence of a function $f : X \to Y$ as well as an inverse to f. For another example, in Section 15.4 we asserted that every functional relation gives rise to a corresponding function, and vice-versa.

What makes these statements interesting is that they involve quantification, both existential and universal, over functions and relations. This takes us outside the realm of first-order logic. One option is to develop a theory in the language of first-order logic in which the universe contains functions, and relations as objects; we will see later that this is what axiomatic set theory does. An alternative is to extend first-order logic to involve new kinds of quantifiers and variables, to range over functions and relations. This is what higher-order logic does.

There are various ways to go about this. In view of the relationship between functions and relations described above, one can take relations as basic, and define functions in terms of them, or vice-versa. The following formulation of higher-order logic, due to the logician Alonzo Church, follows the latter approach. It is sometimes known as *simple type theory*.

Start with some basic types, $X, Y, Z, \ldots$ and a special type, Prop, of propositions. Add the following two rules to build new types:

- If U and V are types, so is $U \times V$.
- If U and V are types, so is $U \to V$.

The first intended to denote the type of ordered pairs (u, v), where u is in U and v is in V. The second is intended to denote the type of functions from U to V. Simple type theory now adds the following means of forming expressions:

- If u is of type U and v is of type V, (u, v) is of type v.
- If p is of type $U \times V$, then $(p)_1$ is of type U and $(p)_2$ if of type V. (These are intended to denote the first and second element of the pair p.)
- If x is a variable of type U, and v is any expression of type V, then $\lambda x\ v$ is of type $U \to V$.
- If f is of type $U \to V$ and u is of type U, $f(u)$ is of type V.

In addition, simple type theory provides all the means we have in first-order logic — boolean connectives, quantifiers, and equality — to build propositions.

A function $f(x, y)$ which takes elements of X and Y to a type Z is viewed as an object of type $X \times Y \to Z$. Similarly, a binary relation $R(x, y)$ on X and Y is viewed as an object of type $X \times Y \to \mathrm{Prop}$. What makes higher-order logic "higher order" is that we can iterate the function type operation indefinitely. For example, if $\mathbb{N}$ is the type of natural numbers, $\mathbb{N} \to \mathbb{N}$ denotes the type of functions from the natural numbers to the natural numbers, and $(\mathbb{N} \to \mathbb{N}) \to \mathbb{N}$ denotes the type of functions $F(f)$ which take a function as argument, and returns a natural number.

We have not specified the syntax and rules of higher-order logic very carefully. This is done in a number of more advanced logic textbooks. The fragment of higher-order logic which allows only functions and relations on the basic types (without iterating these constructions) is known as second-order logic.

These notions should seem familiar; we have been using these constructions, with similar notation, in Lean. Indeed, Lean's logic is an even more elaborate and expressive system of logic, which fully subsumes all the notions of higher-order logic we have discussed here.

16.3 Functions in Lean

The fact that the notions we have been discussing have such a straightforward logical form means that it is easy to define them in Lean. The main difference between the formal representation in Lean and the informal representation above is that, in Lean, we distinguish between a type `X` and a subset `A : set X` of that type.

In Lean's library, composition and identity are defined as follows:

```
variables {X Y Z : Type}

def comp (f : Y → Z) (g : X → Y) : X → Z :=
λx, f (g x)

infixr ` ∘ ` := comp
```

```
def id (x : X) : X :=
x
```

Ordinarily, to use these definitions the notation, you use the command `open function`.

Ordinarily, we use `funext` (for "function extensionality") to prove that two functions are equal.

```
example (f g : X → Y) (h : ∀ x, f x = g x) : f = g :=
funext h
```

But Lean can prove some basic identities by simply unfolding definitions and simplifying expressions, using reflexivity.

```
lemma left_id (f : X → Y) : id ∘ f = f := rfl

lemma right_id (f : X → Y) : f ∘ id = f := rfl

theorem comp.assoc (f : Z → W) (g : Y → Z) (h : X → Y) :
  (f ∘ g) ∘ h = f ∘ (g ∘ h) := rfl

theorem comp.left_id (f : X → Y) : id ∘ f = f := rfl

theorem comp.right_id (f : X → Y) : f ∘ id = f := rfl
```

We can define what it means for f to be injective, surjective, or bijective:

```
def injective (f : X → Y) : Prop := ∀ ⦃x₁ x₂⦄, f x₁ = f x₂ → x₁ = x₂

def surjective (f : X → Y) : Prop := ∀ y, ∃ x, f x = y

def bijective (f : X → Y) := injective f ∧ surjective f
```

Marking the variables `x₁` and `x₂` implicit in the definition of `injective` means that we do not have to write them as often. Specifically, given `h : injective f`, and `h₁ : f x₁ = f x₂`, we write `h h₁` rather than `h x₁ x₂ h₁` to show `x₁ = x₂`.

We can then prove that the identity function is bijective:

```
theorem injective_id : injective (@id X) :=
assume x₁ x₂,
assume H : id x₁ = id x₂,
show x₁ = x₂, from H

theorem surjective_id : surjective (@id X) :=
assume y,
show ∃ x, id x = y, from exists.intro y rfl

theorem bijective_id : bijective (@id X) :=
and.intro injective_id surjective_id
```

More interestingly, we can prove that the composition of injective functions is injective, and so on.

```
theorem injective_comp {g : Y → Z} {f : X → Y}
    (Hg : injective g) (Hf : injective f) :
  injective (g ∘ f) :=
assume x₁ x₂,
assume : (g ∘ f) x₁ = (g ∘ f) x₂,
have f x₁ = f x₂, from Hg this,
show x₁ = x₂, from Hf this

theorem surjective_comp {g : Y → Z} {f : X → Y}
    (hg : surjective g) (hf : surjective f) :
  surjective (g ∘ f) :=
assume z,
exists.elim (hg z) $
assume y (hy : g y = z),
exists.elim (hf y) $
assume x (hx : f x = y),
have g (f x) = z, from eq.subst (eq.symm hx) hy,
show ∃ x, g (f x) = z, from exists.intro x this

theorem bijective_comp {g : Y → Z} {f : X → Y}
    (hg : bijective g) (hf : bijective f) :
  bijective (g ∘ f) :=
have ginj : injective g, from hg.left,
have gsurj : surjective g, from hg.right,
have finj : injective f, from hf.left,
have fsurj : surjective f, from hf.right,
and.intro (injective_comp ginj finj)
  (surjective_comp gsurj fsurj)
```

The notions of left and right inverse are defined in the expected way.

```
-- g is a left inverse to f
def left_inverse (g : Y → X) (f : X → Y) : Prop := ∀ x, g (f x) = x

-- g is a right inverse to f
def right_inverse (g : Y → X) (f : X → Y) : Prop := left_inverse f g
```

In particular, composing with a left or right inverse yields the identity.

```
def id_of_left_inverse {g : Y → X} {f : X → Y} : left_inverse g f → g ∘ f = id :=
assume H, funext H

def id_of_right_inverse {g : Y → X} {f : X → Y} : right_inverse g f → f ∘ g = id :=
assume H, funext H
```

Notice that we need to use `funext` to show the equality of functions.

The following shows that if a function has a left inverse, then it is injective, and if it has a right inverse, then it is surjective.

```
theorem injective_of_left_inverse {g : Y → X} {f : X → Y} :
  left_inverse g f → injective f :=
assume h, assume x₁ x₂, assume feq,
calc x₁ = g (f x₁) : by rw h
    ... = g (f x₂) : by rw feq
    ... = x₂       : by rw h

theorem surjective_of_right_inverse {g : Y  → X} {f : X → Y} :
  right_inverse g f → surjective f :=
assume h, assume y,
let  x : X := g y in
have f x = y, from calc
  f x  = (f (g y))    : rfl
   ... = y            : by rw [h y],
show ∃ x, f x = y, from exists.intro x this
```

16.4 Defining the Inverse Classically

All the theorems listed in the previous section are found in the Lean library, and are available to you when you open the function namespace with `open function`:

```
open function

#check comp
#check left_inverse
#check has_right_inverse
```

Defining inverse functions, however, requires classical reasoning, which we get by opening the classical namespace:

```
open classical

section
  variables A B : Type
  variable P : A → Prop
  variable R : A → B → Prop

  example : (∀ x, ∃ y, R x y) → ∃ f : A → B, ∀ x, R x (f x) :=
  axiom_of_choice

  example (h : ∃ x, P x) : P (some h) :=
  some_spec h
end
```

The axiom of choice tells us that if, for every `x : X`, there is a `y : Y` satisfying `R x y`, then there is a function `f : X → Y` which, for every `x` chooses such a `y`. In Lean, this "axiom" is proved using a classical construction, the `some` function (sometimes called "the indefinite description operator") which, given that there is some `x` satisfying `P x`, returns such an `x`. With these constructions, the inverse function is defined as follows:

```
open classical function
local attribute [instance] prop_decidable

variables {X Y : Type}

noncomputable def inverse (f : X → Y) (default : X) : Y → X :=
λ y, if h : ∃ x, f x = y then some h else default
```

Lean requires us to acknowledge that the definition is not computational, since, first, it may not be algorithmically possible to decide whether or not condition `H` holds, and even if it does, it may not be algorithmically possible to find a suitable value of `x`.

Below, the proposition `inverse_of_exists` asserts that `inverse` meets its specification, and the subsequent theorem shows that if `f` is injective, then the `inverse` function really is a left inverse.

```
theorem inverse_of_exists (f : X → Y) (default : X) (y : Y)
  (h : ∃ x, f x = y) :
f (inverse f default y) = y :=
have h1 : inverse f default y = some h, from dif_pos h,
have h2 : f (some h) = y, from some_spec h,
eq.subst (eq.symm h1) h2

theorem is_left_inverse_of_injective (f : X → Y) (default : X)
  (injf : injective f) :
left_inverse (inverse f default) f :=
let finv := (inverse f default) in
assume x,
have h1 : ∃ x', f x' = f x, from exists.intro x rfl,
have h2 : f (finv (f x)) = f x, from inverse_of_exists f default (f x) h1,
show finv (f x) = x, from injf h2
```

16.5 Functions and Sets in Lean

In Section 7.4 we saw how to represent relativized universal and existential quantifiers when formalizing phrases like "every prime number greater than two is odd" and "some prime number is even." In a similar way, we can relativize statements to sets. In symbolic logic, the expression $\exists x \in A \; P(x)$ abbreviates $\exists x \; (x \in A \wedge P(x))$, and $\forall x \in A \; P(x)$ abbreviates $\forall x \; (x \in A \to P(x))$.

Lean also defines notation for relativized quantifiers:

```
variables (X : Type) (A : set X) (P : X → Prop)

#check ∀ x ∈ A, P x
#check ∃ x ∈ A, P x
```

Here is an example of how to use the bounded universal quantifier:

```
example (h : ∀ x ∈ A, P x) (x : X) (h1 : x ∈ A) : P x := h x h1
```

Using bounded quantifiers, we can talk about the behavior of functions on particular sets:

```
import data.set
open set function

variables {X Y : Type}
variables (A  : set X) (B : set Y)

def maps_to (f : X → Y) (A : set X) (B : set Y) := ∀ x ∈ A, f x ∈ B

def inj_on (f : X → Y) (A : set X) := ∀ (x₁ ∈ A) (x₂ ∈ A), f x₁ = f x₂ → x₁ = x₂

def surj_on (f : X → Y) (A : set X) (B : set Y) := B ⊆ f '' A
```

The expression `maps_to f A B` asserts that `f` maps elements of the set `A` to the set `B`, and the expression `inj_on f A` asserts that `f` is injective on `A`. The expression `surj_on f A B` asserts that, viewed as a function defined on elements of `A`, the function `f` is surjective onto the set `B`. Here are examples of how they can be used:

```
variables (f : X → Y) (A : set X) (B : set Y)

example (h : maps_to f A B) (x : X) (h1 : x ∈ A) : f x ∈ B := h x h1

example (h : inj_on f A) (x₁ x₂ : X) (h1 : x₁ ∈ A) (h2 : x₂ ∈ A)
    (h3 : f x₁ = f x₂) : x₁ = x₂ :=
h x₁ h1 x₂ h2 h3

example (h : surj_on f A B) (y : Y) (h1 : y ∈ B) : ∃ x, x ∈ A ∧ f x = y :=
h h1
```

Actually, it is slightly more convenient to mark the variables in `maps_to` and `inj_on` as implicit:

```
def maps_to (f : X → Y) (A : set X) (B : set Y) := ∀ {x}, x ∈ A → f x ∈ B

def inj_on (f : X → Y) (A : set X) := ∀ {x₁ x₂}, x₁ ∈ A → x₂ ∈ A → f x₁ = f x₂ → x₁ = x₂
```

In that case, we can leave out some arguments:

```
variables (f : X → Y) (A : set X) (B : set Y)

example (h : maps_to f A B) (x : X) (h1 : x ∈ A) : f x ∈ B := h h1

example (h : inj_on f A) (x₁ x₂ : X) (h1 : x₁ ∈ A) (h2 : x₂ ∈ A)
    (h3 : f x₁ = f x₂) : x₁ = x₂ :=
h h1 h2 h3
```

In the examples below, we'll use the versions with implicit arguments.

The expression `surj_on f A B` asserts that, viewed as a function defined on elements of `A`, the function `f` is surjective onto the set `B`:

With these notions in hand, we can prove that the composition of injective functions is injective. The proof is similar to the one above, though now we have to be more careful to relativize claims to `A` and `B`:

```
theorem inj_on_comp (fAB : maps_to f A B) (hg : inj_on g B) (hf: inj_on f A) :
  inj_on (g ∘ f) A :=
assume x1 x2 : X,
assume x1A : x1 ∈ A,
assume x2A : x2 ∈ A,
have fx1B : f x1 ∈ B, from fAB x1A,
have fx2B : f x2 ∈ B, from fAB x2A,
assume h1 : g (f x1) = g (f x2),
have h2 : f x1 = f x2, from hg fx1B fx2B h1,
show x1 = x2, from hf x1A x2A h2
```

We can similarly prove that the composition of surjective functions is surjective:

```
theorem surj_on_comp (hg : surj_on g B C) (hf: surj_on f A B) :
  surj_on (g ∘ f) A C :=
assume z,
assume zc : z ∈ C,
exists.elim (hg zc) $
assume y (h1 : y ∈ B ∧ g y = z),
exists.elim (hf (and.left h1)) $
assume x (h2 : x ∈ A ∧ f x = y),
show ∃x, x ∈ A ∧ g (f x) = z, from
  exists.intro x
    (and.intro
      (and.left h2)
      (calc
        g (f x) = g y : by rw and.right h2
            ... = z   : by rw and.right h1))
```

The following shows that the image of a union is the union of images:

```
theorem image_union : f '' (A₁ ∪ A₂) =f '' A₁ ∪ f '' A₂ :=
ext (assume y, iff.intro
  (assume h : y ∈ image f (A₁ ∪ A₂),
    exists.elim h $
    assume x h1,
    have xA₁A₂ : x ∈ A₁ ∪ A₂, from h1.left,
    have fxy : f x = y, from h1.right,
    or.elim xA₁A₂
      (assume xA₁, or.inl (mem_image xA₁ fxy))
      (assume xA₂, or.inr (mem_image xA₂ fxy)))
  (assume h : y ∈ image f A₁ ∪ image f A₂,
    or.elim h
      (assume yifA₁ : y ∈ image f A₁,
        exists.elim yifA₁ $
        assume x h1,
        have xA₁ : x ∈ A₁, from h1.left,
        have fxy : f x = y, from h1.right,
        mem_image (or.inl xA₁) fxy)
      (assume yifA₂ : y ∈ image f A₂,
        exists.elim yifA₂ $
        assume x h1,
        have xA₂ : x ∈ A₂, from h1.left,
        have fxy : f x = y, from h1.right,
        mem_image (or.inr xA₂) fxy)))
```

16.6 Exercises

1. Fill in the sorry's in the last three proofs below.

```
open function int algebra

def f (x : ℤ) : ℤ := x + 3
def g (x : ℤ) : ℤ := -x
def h (x : ℤ) : ℤ := 2 * x + 3

example : injective f :=
assume x1 x2,
assume h1 : x1 + 3 = x2 + 3,   -- Lean knows this is the same as f x1 = f x2
show x1 = x2, from eq_of_add_eq_add_right h1

example : surjective f :=
assume y,
have h1 : f (y - 3) = y, from calc
  f (y - 3) = (y - 3) + 3 : rfl
        ... = y           : by rw sub_add_cancel,
show ∃ x, f x = y, from exists.intro (y - 3) h1

example (x y : ℤ) (h : 2 * x = 2 * y) : x = y :=
have h1 : 2 ≠ (0 : ℤ), from dec_trivial,  -- this tells Lean to figure it out itself
show x = y, from eq_of_mul_eq_mul_left h1 h

example (x : ℤ) : -(-x) = x := neg_neg x

example (A B : Type) (u : A → B) (v : B → A) (h : left_inverse u v) :
  ∀ x, u (v x) = x :=
h

example (A B : Type) (u : A → B) (v : B → A) (h : left_inverse u v) :
  right_inverse v u :=
h

-- fill in the sorry's in the following proofs

example : injective h :=
sorry

example : surjective g :=
sorry

example (A B : Type) (u : A → B) (v1 : B → A) (v2 : B → A)
  (h1 : left_inverse v1 u) (h2 : right_inverse v2 u) : v1 = v2 :=
funext
  (assume x,
    calc
      v1 x = v1 (u (v2 x)) : sorry
       ... = v2 x           : sorry)
```

2. Fill in the sorry in the proof below.

```
import data.set
open function set

variables {X Y : Type}
variable  f : X → Y
variables A B : set X

example : f '' (A ∪ B) = f '' A ∪ f '' B :=
eq_of_subset_of_subset
  (assume y,
    assume h1 : y ∈ f '' (A ∪ B),
    exists.elim h1 $
    assume x h,
    have h2 : x ∈ A ∪ B, from h.left,
    have h3 : f x = y, from h.right,
    or.elim h2
      (assume h4 : x ∈ A,
        have h5 : y ∈ f '' A, from mem_image h4 h3,
        show y ∈ f '' A ∪ f '' B, from or.inl h5)
      (assume h4 : x ∈ B,
        have h5 : y ∈ f ''  B, from mem_image h4 h3,
        show y ∈ f '' A ∪ f '' B, from or.inr h5))
  (assume y,
    assume h2 : y ∈ f '' A ∪ f '' B,
    or.elim h2
      (assume h3 : y ∈ f '' A,
        exists.elim h3 $
        assume x h,
        have h4 : x ∈ A, from h.left,
        have h5 : f x = y, from h.right,
        have h6 : x ∈ A ∪ B, from or.inl h4,
        show y ∈ f '' (A ∪ B), from mem_image h6 h5)
      (assume h3 : y ∈ f '' B,
        exists.elim h3 $
        assume x h,
        have h4 : x ∈ B, from h.left,
        have h5 : f x = y, from h.right,
        have h6 : x ∈ A ∪ B, from or.inr h4,
        show y ∈ f '' (A ∪ B), from mem_image h6 h5))

-- remember, x ∈ A ∩ B is the same as x ∈ A ∧ x ∈ B
example (x : X) (h1 : x ∈ A) (h2 : x ∈ B) : x ∈ A ∩ B :=
and.intro h1 h2

example (x : X) (h1 : x ∈ A ∩ B) : x ∈ A :=
and.left h1

-- Fill in the proof below.
-- (It should take about 8 lines.)

example : f '' (A ∩ B) ⊆ f '' A ∩ f '' B :=
assume y,
assume h1 : y ∈ f '' (A ∩ B),
show y ∈ f '' A ∩ f '' B, from sorry
```

CHAPTER SEVENTEEN

THE NATURAL NUMBERS AND INDUCTION

This chapter marks a transition from the abstract to the concrete. Viewing the mathematical universe in terms of sets, relations, and functions gives us useful ways of thinking about mathematical objects and structures and the relationships between them. At some point, however, we need to start thinking about *particular* mathematical objects and structures, and the natural numbers are a good place to start. The nineteenth century mathematician Leopold Kronecker once proclaimed "God created the whole numbers; everything else is the work of man." By this he meant that the natural numbers (and the integers, which we will also discuss below) are a fundamental component of the mathematical universe, and that many other objects and structures of interest can be constructed from these.

In this chapter, we will consider the natural numbers and the basic principles that govern them. In Chapter 18 we will see that even basic operations like addition and multiplication can be defined using means described here, and their properties derived from these basic principles. Our presentation in this chapter will remain informal, however. In Chapter 19, we will see how these principles play out in number theory, one of the oldest and most venerable branches of mathematics.

17.1 The Principle of Induction

The set of natural numbers is the set

In the past, opinions have differed as to whether the set of natural numbers should start with 0 or 1, but these days most mathematicians take them to start with 0. Logicians often call the function $s(n) = n + 1$ the *successor* function, since it maps each natural number, n, to the one that follows it. What makes the natural numbers special is that they are *generated* by the number zero and the successor function, which is to say, the only way to "construct" a natural number is to start with 0 and apply the successor function finitely many times. From a foundational standpoint, we are in danger of running into a circularity here, because it is not clear how we can explain what it means to apply a function "finitely many times" without talking about the natural numbers themselves. But the following principle, known as the *principle of induction*, describes this essential property of the natural numbers in a non-circular way.

Principle of Induction. Let P be any property of natural numbers. Suppose P holds of zero, and whenever P holds of a natural number n, then it holds of its successor, $n + 1$. Then P holds of every natural number.

This reflects the image of the natural numbers as being generated by zero and the successor operation: by covering the zero and successor cases, we take care of all the natural numbers.

The principle of induction provides a recipe for proving that every natural number has a certain property: to show that P holds of every natural number, show that it holds of 0, and show that whenever it holds of some number n, it holds of $n + 1$. This form of proof is called a *proof by induction*. The first required task is called the *base case*, and the second required task is called the *induction step*. The induction step requires temporarily fixing a natural number n, assuming that P holds of n, and then showing that P holds of $n+1$. In this context, the assumption that P holds of n is called the *inductive hypothesis*.

You can visualize proof by induction as a method of knocking down an infinite stream of dominoes, all at once. We set the mechanism in place and knock down domino 0 (the base case), and every domino knocks down the next domino (the induction step). So domino 0 knocks down domino 1; that knocks down domino 2, and so on.

Here is an example of a proof by induction.

Theorem. For every natural number n,

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

Proof. We prove this by induction on n. In the base case, when $n = 0$, we have $1 = 2^{0+1} - 1$, as required.

For the induction step, fix n, and assume

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

(the induction hypothesis). We need to show that this same claim holds with n replaced by $n+1$. But this is just a calculation:

$$\begin{aligned} 1 + 2 + \ldots + 2^{n+1} &= (1 + 2 + \ldots + 2^n) + 2^{n+1} \\ &= 2^{n+1} - 1 + 2^{n+1} \\ &= 2 \cdot 2^{n+1} - 1 \\ &= 2^{n+2} - 1. \end{aligned}$$

In the notation of first-order logic, if we write $P(n)$ to mean that P holds of n, we could express the principle of induction as follows:

$$P(0) \wedge \forall n\ (P(n) \to P(n+1)) \to \forall n\ P(n).$$

But notice that the principle of induction says that the axiom holds *for every property* P, which means that we should properly use a universal quantifier for that, too:

$$\forall P\ (P(0) \wedge \forall n\ (P(n) \to P(n+1)) \to \forall n\ P(n)).$$

Quantifying over properties takes us out of the realm of first-order logic; induction is therefore a second-order principle.

The pattern for a proof by induction is expressed even more naturally by the following natural deduction rule:

$$\dfrac{P(0) \qquad \begin{array}{c} \overline{P(n)}^{\,1} \\ \vdots \\ P(n+1) \end{array}}{\forall n\ P(n)}$$

You should think about how some of the proofs in this chapter could be represented formally using natural deduction.

For another example of a proof by induction, let us derive a formula that, given any finite set S, determines the number of subsets of S. For example, there are four subsets of the two-element set $\{1, 2\}$, namely $\emptyset$, $\{1\}$, $\{2\}$, and $\{1, 2\}$. You should convince yourself that there are eight subsets of the set $\{1, 2, 3\}$. The following theorem establishes the general pattern.

Theorem. For any finite set S, if S has n elements, then there are 2^n subsets of S.

Proof. We use induction on n. In the base case, there is only one set with 0 elements, the empty set, and there is exactly one subset of the empty set, as required.

In the inductive case, suppose S has $n + 1$ elements. Let a be any element of S, and let S' be the set containing the remaining n elements. In order to count the subsets of S, we divide them into two groups.

First, we consider the subsets of S that don't contain a. These are exactly the subsets of S', and by the inductive hypothesis, there are 2^n of those.

Next we consider the subsets of S that *do* contain a. Each of these is obtained by choosing a subset of S' and adding a. Since there are 2^n subsets of S', there are 2^n subsets of S that contain a.

Taken together, then, there are $2^n + 2^n = 2^{n+1}$ subsets of S, as required.

We have seen that there is a correspondence between properties of a domain and subsets of a domain. For every property P of natural numbers, we can consider the set S of natural numbers with that property, and for every set of natural numbers, we can consider the property of being in that set. For example, we can talk about the property of being even, or talk about the set of even numbers. Under this correspondence, the principle of induction can be cast as follows:

Principle of Induction. Let S be any set of natural numbers that contains 0 and is closed under the successor operation. Then $S = \mathbb{N}$.

Here, saying that S is "closed under the successor operation" means that whenever a number n is in S, so is $n + 1$.

17.2 Variants of Induction

In this section, we will consider variations on the principle of induction that are often useful. It is important to recognize that each of these can be justified using the principle of induction as stated in the last section, so they need not be taken as fundamental.

The first one is no great shakes: instead of starting from 0, we can start from any natural number, m.

Principle of Induction from a Starting Point. Let P be any property of natural numbers, and let m be any natural number. Suppose P holds of m, and whenever P holds of a natural number n greater than or equal to m, then it holds of its successor, $n + 1$. Then P holds of every natural number greater than or equal to m.

Assuming the hypotheses of this last principle, if we let $P'(n)$ be the property "P holds of $m + n$," we can prove that P' holds of every n by the ordinary principle of induction. But this means that P holds of every number greater than or equal to m.

Here is one example of a proof using this variant of induction.

Theorem. For every natural number $n \geq 5$, $2^n > n^2$.

Proof. By induction on n. When $n = 5$, we have $2^n = 32 > 25 = n^2$, as required.

For the induction step, suppose $n \geq 5$ and $2^n > n^2$. Since n is greater than or equal to 5, we have $2n + 1 \leq 3n \leq n^2$, and so

$$\begin{aligned}(n+1)^2 &= n^2 + 2n + 1 \\ &\leq n^2 + n^2 \\ &< 2^n + 2^n \\ &= 2^{n+1}.\end{aligned}$$

For another example, let us derive a formula for the sum total of the angles in a convex polygon. A polygon is said to be *convex* if every line between two vertices stays inside the polygon. We will accept without proof the visually obvious fact that one can subdivide any convex polygon with more than three sides into a triangle and a convex polygon with one fewer side, namely, by closing off any two consecutive sides to form a triangle. We will also accept, without proof, the basic geometric fact that the sum of the angles of any triangle is 180 degrees.

Theorem. For any $n \geq 3$, the sum of the angles of any convex n-gon is $180(n - 2)$.

Proof. In the base case, when $n = 3$, this reduces to the statement that the sum of the angles in any triangle is 180 degrees.

For the induction step, suppose $n \geq 3$, and let P be a convex $(n + 1)$-gon. Divide P into a triangle and an n-gon. By the inductive hypotheses, the sum of the angles of the n-gon is $180(n - 2)$ degrees, and the sum of the angles of the triangle is 180 degrees. The measures of these angles taken together make up the sum of the measures of the angles of P, for a total of $180(n - 2) + 180 = 180(n - 1)$ degrees.

For our second example, we will consider the principle of *complete induction*, also sometimes known as *total induction.*

Principle of Complete Induction. Let P be any property that satisfies the following: for any natural number n, whenever P holds of every number less than n, it also holds of n. Then P holds of every natural number.

Notice that there is no need to break out a special case for zero: for any property P, P holds of all the natural numbers less than zero, for the trivial reason that there aren't any! So, in particular, any such property automatically holds of zero.

Notice also that if such a property P holds of every number less than n, then it also holds of every number less than $n + 1$ (why?). So, for such a P, the ordinary principle of induction implies that for every natural number n, P holds of every natural number less than n. But this is just a roundabout way of saying that P holds of every natural number. In other words, we have justified the principle of complete induction using ordinary induction.

To use the principle of complete induction we merely have to let n be any natural number and show that P holds of n, assuming that it holds of every smaller number. Compare this to the ordinary principle of induction, which requires us to show $P(n+1)$ assuming only $P(n)$. The following example of the use of this principle is taken verbatim from the introduction to this book:

Theorem. Every natural number greater than or equal to 2 can be written as a product of primes.

Proof. We proceed by induction on n. Let n be any natural number greater than 2. If n is prime, we are done; we can consider n itself as a product with one factor. Otherwise, n is composite, and we can write $n = m \cdot k$ where m and k are smaller than n and greater than 1. By the inductive hypothesis, each of m and k can be written as a product of primes, say

$$m = p_1 \cdot p_2 \cdot \ldots \cdot p_u$$

and

$$k = q_1 \cdot q_2 \cdot \ldots \cdot q_v.$$

But then we have

$$n = m \cdot k = p_1 \cdot p_2 \cdot \ldots \cdot p_u \cdot q_1 \cdot q_2 \cdot \ldots \cdot q_v,$$

a product of primes, as required.

Finally, we will consider another formulation of induction, known as the least element principle.

The Least Element Principle. Suppose P is some property of natural numbers, and suppose P holds of some n. Then there is a smallest value of n for which P holds.

In fact, using classical reasoning, this is equivalent to the principle of complete induction. To see this, consider the contrapositive of the statement above: "if there is no smallest value for which P holds, then P doesn't hold of any natural number." Let $Q(n)$ be the property P does *not* hold of n. Saying that there is no smallest value for which P holds means that, for every n, if P holds at n, then it holds of some number smaller than n; and this is equivalent to saying that, for every n, if Q doesn't hold at n, then there is a smaller value for which Q doesn't hold. And *that* is equivalent to saying that if Q holds for every number less than n, it holds for n as well. Similarly, saying that P doesn't hold of any natural number is equivalent to saying that Q holds of every natural number. In other words, replacing the least element principle by its contrapositive, and replacing P by "not Q," we have the principle of complete induction. Since every statement is equivalent to its contrapositive, and every predicate as its negated version, the two principles are the same.

It is not surprising, then, that the least element principle can be used in much the same way as the principle of complete induction. Here, for example, is a formulation of the previous proof in these terms. Notice that it is phrased as a proof by contradiction.

Theorem. Every natural number greater than equal to 2 can be written as a product of primes.

Proof. Suppose, to the contrary, there some natural number greater than or equal to 2 cannot be written as a product of primes. By the least element principle, there is a smallest such element; call it n. Then n is not prime, and since it is greater than or equal to 2, it must be composite. Hence we can write $n = m \cdot k$ where m and k are smaller than n and greater than 1. By the assumption on n, each of m and k can be written as a product of primes, say

$$m = p_1 \cdot p_2 \cdot \ldots \cdot p_u$$

and

$$k = q_1 \cdot q_2 \cdot \ldots \cdot q_v.$$

But then we have

$$n = m \cdot k = p_1 \cdot p_2 \cdot \ldots \cdot p_u \cdot q_1 \cdot q_2 \cdot \ldots \cdot q_v,$$

a product of primes, contradicting the fact that n cannot be written as a product of primes.

Here is another example:

Theorem. Every natural number is interesting.

Proof. Suppose, to the contrary, some natural number is uninteresting. Then there is a smallest one, n. In other words, n is the smallest uninteresting number. But that is really interesting! Contradiction.

17.3 Recursive Definitions

Suppose I tell you that I have a function $f : \mathbb{N} \to \mathbb{N}$ in mind, satisfying the following properties:

$$f(0) = 1$$
$$f(n+1) = 2 \cdot f(n)$$

What can you infer about f? Try calculating a few values:

$$f(1) = f(0+1) = 2 \cdot f(0) = 2$$
$$f(2) = f(1+1) = 2 \cdot f(1) = 4$$
$$f(3) = f(2+1) = 2 \cdot f(2) = 8$$

It soon becomes apparent that for every n, $f(n) = 2^n$.

What is more interesting is that the two conditions above specify *all* the values of f, which is to say, there is exactly one function meeting the specification above. In fact, it does not matter that f takes values in the natural numbers; it could take values in any other domain. All that is needed is a value of $f(0)$ and a way to compute the value of $f(n+1)$ in terms of n and $f(n)$. This is what the principle of definition by recursion asserts:

Principle of Definition by Recursion. Let A be any set, and suppose a is in A, and $g : \mathbb{N} \times A \to A$. Then there is a unique function f satisfying the following two clauses:

$$f(0) = a$$
$$f(n+1) = g(n, f(n)).$$

The principle of recursive definition makes two claims at once: first, that there is a function f satisfying the clauses above, and, second, that any two functions f_1 and f_2 satisfying those clauses are equal, which is to say, they have the same values for every input. In the example with which we began this section, A is just $\mathbb{N}$ and $g(n, f(n)) = 2 \cdot f(n)$.

In some axiomatic frameworks, the principle of recursive definition can be justified using the principle of induction. In others, the principle of induction can be viewed as a special case of the principle of recursive definition. For now, we will simply take both to be fundamental properties of the natural numbers.

As another example of a recursive definition, consider the function $g : \mathbb{N} \to \mathbb{N}$ defined recursively by the following clauses:

$$\begin{aligned} g(0) &= 1 \\ g(n+1) &= (n+1) \cdot g(n) \end{aligned}$$

Try calculating the first few values. Unwrapping the definition, we see that $g(n) = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot (n-1) \cdot n$ for every n; indeed, definition by recursion is usually the proper way to make expressions using "…" precise. The value $g(n)$ is read "n factorial," and written $n!$.

Indeed, summation notation

$$\sum_{i<n} f(i) = f(0) + f(1) + \ldots + f(n-1)$$

$$andproductnotation$$

$$\prod_{i<n} f(i) = f(0) \cdot f(1) \cdot \cdots \cdot f(n-1)$$

can also be made precise using recursive definitions. For example, the function $k(n) = \sum_{i<n} f(i)$ can be defined recursively as follows:

$$\begin{aligned} k(0) &= 0 \\ k(n+1) &= k(n) + f(n) \end{aligned}$$

Induction and recursion are complementary principles, and typically the way to prove something about a recursively defined function is to use the principle of induction. For example, the following theorem provides a formulas for the sum $1 + 2 + \ldots + n$, in terms of n.

Theorem. For every n, $\sum_{i<n+1} i = n(n+1)/2$.

Proof. In the base case, when $n = 0$, both sides are equal to 0.

In the inductive step, we have

$$\begin{aligned} \sum_{i<n+2} i &= \left(\sum_{i<n+1} i\right) + (n+1) \\ &= n(n+1)/2 + n + 1 \\ &= \frac{n^2+n}{2} + \frac{2n+2}{2} \\ &= \frac{n^2+3n+2}{2} \\ &= \frac{(n+1)(n+2)}{2}. \end{aligned}$$

There are just as many variations on the principle of recursive definition as there are on the principle of induction. For example, in analogy to the principle of complete induction, we can specify a value of $f(n)$ in terms of the values that f takes at all inputs smaller than n. When $n \geq 2$, for example, the following definition specifies that value of a function $\mathrm{fib}(n)$ in terms of its two predecessors:

$$\begin{aligned}\mathrm{fib}(0) &= 0\\ \mathrm{fib}(1) &= 1\\ \mathrm{fib}(n+2) &= \mathrm{fib}(n+1) + \mathrm{fib}(n).\end{aligned}$$

Calculating the values of fib on $0, 1, 2, \ldots$ we obtain

$$0, 1, 1, 2, 3, 5, 8, 13, 21, \ldots$$

Here, after the second number, each successive number is the sum of the two values preceding it. This is known as the *Fibonacci sequence*, and the corresponding numbers are known as the *Fibonacci numbers*. An ordinary mathematical presentation would write F_n instead of $\mathrm{fib}(n)$ and specify the sequence with the following equations:

$$F_0 = 0, \quad F_1 = 1, \quad F_{n+2} = F_{n+1} + F_n$$

But you can now recognize such a specification as an implicit appeal to the principle of definition by recursion. We ask you to prove some facts about the Fibonacci sequence in the exercises below.

17.4 Arithmetic on the Natural Numbers

In the next chapter, we will see that it is even possible to define addition and multiplication recursively, and to establish most of their basic properties using the principle of recursion. This is important from a foundational perspective, in which, as much as possible, we want to ground our reasoning on a small number of fundamental principles. Just as the foundations of a building are below ground, however, the foundations of mathematics should only be visible when we choose to go down to the basement and look around. In this section, we summarize the basic properties of natural numbers that play a role in day-to-day mathematics. In an ordinary mathematical argument or calculation, they can be used without explicit justification.

$$\begin{aligned}m + n &= n + m \quad \text{(commutativity of addition)}\\ m + (n + k) &= (m + n) + k \quad \text{(associativity of addition)}\\ n + 0 &= n \quad \text{(0 is a neutral element for addition)}\\ n \cdot m &= m \cdot n \quad \text{(commutativity of multiplication)}\\ m \cdot (n \cdot k) &= (m \cdot n) \cdot k \quad \text{(associativity of multiplication)}\\ n \cdot 1 &= n \quad \text{(1 is an neutral element for multiplication)}\\ n \cdot (m + k) &= n \cdot m + n \cdot k \quad \text{(distributivity)}\\ n \cdot 0 &= 0 \quad \text{(0 is an absorbing element for multiplication)}\end{aligned}$$

We also have the following properties:

- $n + 1 \neq 0$;
- if $n + k = m + k$ then $n = m$;
- if $n \cdot k = m \cdot k$ and $k \neq 0$ then $n = m$.

We can define $m \leq n$, "m is less than or equal to n," to mean that there exists a k such that $m + k = n$. If we do that, it is not hard to show that the less-than-or-equal-to relation satisfies all the following properties, for every n, m, and k:

- $n \leq n$ (*reflexivity*);
- if $n \leq m$ and $m \leq k$ then $n \leq k$ (*transitivity*);
- if $n \leq m$ and $m \leq n$ then $n = m$ (*antisymmetry*);
- for all n and m, either $n \leq m$ or $m \leq n$ is true (*totality*);
- if $n \leq m$ then $n + k \leq m + k$;
- if $n \leq m$ then $nk \leq mk$;
- if $m \geq n$ then $m = n$ or $m \geq n + 1$;
- $0 \leq n$.

Remember from Chapter 13 that the first four items assert that $\leq$ is a linear order. Note that when we write $m \geq n$, we mean $n \leq m$.

We can then define $m < n$, "m is less than n," to mean $m + 1 \leq n$. The following proposition then justifies the terminology.

Proposition. With the definitions above, for every m and n, $m \leq n$ if and only if $m < n$ or $m = n$.

Proof. First, suppose $m \leq n$, and let us show $m < n$ or $m = n$. Since $m \leq n$, then $m + k = n$. If $k = 0$, we have $m = n$. Otherwise, $k \geq 1$, and we have $m + 1 \leq m + k = n$, which mean $m < n$.

Conversely, suppose $m < n$ or $m = n$. If $m < n$, then we have $m \leq m + 1 \leq n$, so $m \leq n$. And if $m = n$, we also have $m \leq n$, as required.

In a similar way, we can show that $m < n$ if and only if $m \leq n$ and $m \neq n$. In fact, we can demonstrate all of the following from these properties and the properties of $\leq$:

- $n < n$ is never true (*irreflexivity*);
- if $n < m$ and $m < k$ then $n < k$ (*transitivity*);
- for all n and m, either $n < m$, $n = m$ or $m < n$ is true (*trichotomy*);
- if $n < m$ then $n + k < m + k$;
- if $k > 0$ and $n < m$ then $nk < mk$;
- if $m > n$ then $m = n + 1$ or $m > n + 1$;
- for all n, $n = 0$ or $n > 0$.

The first three items mean that $<$ is a strict linear order, and the properties above means that $\leq$ is the associated linear order, in the sense described in Section 13.1.

Proof. We will prove some of these properties.

The first property is straightforward: we know $n \leq n+1$, and if we had $n+1 \leq n$, we should have $n = n+1$, a contradiction.

For the second property, assume $n < m$ and $m < k$. Then $n + 1 \leq m \leq m + 1 \leq k$, which implies $n < k$.

For the third, we know that either $n \leq m$ or $m \leq n$. If $m = n$, we are done, and otherwise we have either $n < m$ or $m < n$.

For the fourth, if $n + 1 \leq m$, we have $n + 1 + k = (n + k) + 1 \leq m + k$, as required.

For the fifth, suppose $k > 0$, which is to say, $k \geq 1$. If $n < m$, then $n + 1 \leq m$, and so $nk + 1 \leq nk + k \leq mk$. But this implies $nk < mk$, as required.

The rest of the remaining proofs are left as an exercise to the reader.

Here are some additional properties of $<$ and $\leq$:

- $n < m$ and $m < n$ cannot both hold (*asymmetry*);
- $n + 1 > n$;
- if $n < m$ and $m \leq k$ then $n < k$;
- if $n \leq m$ and $m < k$ then $n < k$;
- if $m > n$ then $m \geq n + 1$;
- if $m \geq n$ then $m + 1 > n$;
- if $n + k < m + k$ then $n < m$;
- if $nk < mk$ then $k > 0$ and $n < m$.

These can be proved from the ones above. Moreover, the collection of principles we have just seen can be used to justify basic facts about the natural numbers, which are again typically taken for granted in informal mathematical arguments.

Proposition. If n and m are natural numbers such that $n + m = 0$, then $n = m = 0$.

Proof. We first prove that $m = 0$. We know that $m = 0$ or $m > 0$. Suppose that $m > 0$. Then $n + m > n + 0 = n$. Since $n \geq 0$, we conclude that $n + m > 0$, which contradicts the fact that $n + m = 0$. Since $m > 0$ leads to a contradiction, we must have $m = 0$.

Now we can easily conclude that $n = 0$, since $n = n + 0 = n + m = 0$. Hence $n = m = 0$.

Proposition. If n is a natural number such that $n < 3$, then $n = 0$, $n = 1$ or $n = 2$.

Proof. In this proof we repeatedly use the property that if $m > n$ then $m = n + 1$ or $m > n + 1$. Since $2 + 1 = 3 > n$, we conclude that either $2 + 1 = n + 1$ or $2 + 1 > n + 1$. In the first case we conclude $n = 2$, and we are done. In the second case we conclude $2 > n$, which implies that either $2 = n + 1$, or $2 > n + 1$. In the first case, we conclude $n = 1$, and we are done. In the second case, we conclude $1 > n$, and appeal one last time to the general principle presented above to conclude that either $1 = n + 1$ or $1 > n + 1$. In the first case, we conclude $n = 0$, and we are once again done. In the second case, we conclude that $0 > n$. This leads to a contradiction, since now $0 > n \geq 0$, hence $0 > 0$, which contradicts the irreflexivity of $>$.

17.5 The Integers

The natural numbers are designed for counting discrete quantities, but they suffer an annoying drawback: it is possible to subtract n from m if n is less than or equal to m, but not if m is greater than n. The set of *integers*, $\mathbb{Z}$, extends the natural numbers with negative values, to make it possible to carry out subtraction in full:

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

We will see in a later chapter that the integers can be extended to the *rational numbers*, the *real numbers*, and the *complex numbers*, each of which serves useful purposes. For dealing with discrete quantities, however, the integers will get us pretty far.

You can think of the integers as consisting of two copies of the natural numbers, a positive one and a negative one, sharing a common zero. Conversely, once we have the integers, you can think of the natural numbers as consisting of the nonnegative integers, that is, the integers that are greater than or equal to 0. Most mathematicians blur the distinction between the two, though we will see that in Lean, for example, the natural numbers and the integers represent two different data types.

Most of the properties of the natural numbers that were enumerated in the last section hold of the integers as well, but not all. For example, it is no longer the case that $n + 1 \neq 0$ for every n, since the claim is false for $n = -1$. For another example, it is not the case that every integer is either equal to 0 or greater than 0, since this fails to hold of the negative integers.

The key property that the integers enjoy, which sets them apart from the natural numbers, is that for every integer n there is a value $-n$ with the property that $n + (-n) = 0$. The value $-n$ is called the *negation* of n. We define subtraction $n - m$ to be $n + (-m)$. For any integer n, we also define the *absolute value* of n, written $|n|$, to be n if $n \geq 0$, and $-n$ otherwise.

We can no longer use proof by induction on the integers, because induction does not cover the negative numbers. But we can use induction to show that a property holds of every nonnegative integer, for example. Moreover, we know that every negative integer is the negation of a positive one.As a result, proofs involving the integers often break down into two cases, where one case covers the nonnegative integers, and the other case covers the negative ones.

17.6 Exercises

1. Write the principle of complete induction using the notation of symbolic logic. Also write the least element principle this way, and use logical manipulations to show that the two are equivalent.
2. Show that for every n, $0^2 + 1^2 + 2^2 + \ldots n^2 = \frac{1}{6}n(n+1)(n+2)$.
3. Show that for every n, $0^3 + 1^3 + \ldots + n^3 = \frac{1}{4}n^2(n+1)^2$.
4. Given the definition of the Fibonacci numbers in Section 17.3, prove Cassini's identity: for every n, $F_{n+1}^2 - F_{n+2}F_n = (-1)^n$. Hint: in the induction step, write F_{n+2}^2 as $F_{n+2}(F_{n+1} + F_n)$.
5. Prove $\sum_{i<n} F_{2i+1} = F_{2n}$.
6. Prove the following two identities:
   - $F_{2n+1} = F_{n+1}^2 + F_n^2$
   - $F_{2n+2} = F_{n+2}^2 - F_n^2$

   Hint: use induction on n, and prove them both at once. In the induction step, expand $F_{2n+3} = F_{2n+2} + F_{2n+1}$, and similarly for F_{2n+4}. Proving the second equation is especially tricky. Use the inductive hypothesis and the first identity to simplify the left-hand side, and repeatedly unfold the Fibonacci number with the highest index and simplify the equation you need to prove. (When you have worked out a solution, write a clear equational proof, calculating in the "forwards" direction.)
7. Prove that every natural number can be written as a sum of *distinct* powers of 2. For this problem, $1 = 2^0$ is counted as power of 2.
8. Let V be a non-empty set of integers such that the following two properties hold:
   - if $x, y \in V$, then $x - y \in V$
   - if $x \in V$, then every multiple of x is an element of V

Prove that there is some $d \in V$, such that V is equal to the set of multiples of d. Hint: use the least element principle.

9. Following the example in Section 17.4, prove that if n is a natural number and $n < 5$, then n is one of the values $0, 1, 2, 3$, or 4.
10. Prove that if n and m are natural numbers and $nm = 1$, then $n = m = 1$.

    This is tricky. First show that n and m are greater than 0, and hence greater than or equal to 1. Then show that if either one of them is greater than 1, then $nm > 1$.
11. Prove all the claims in Section 17.4 that were stated without proof.
12. Prove the following properties of negation and subtraction on the integers, using only the properties of negation and subtraction given in Section 17.5.
    - If $n + m = 0$ then $m = -n$;
    - $-0 = 0$;
    - If $-n = -m$ then $n = m$;
    - $m + (n - m) = n$;
    - $-(n + m) = -n - m$;
    - If $m < n$ then $n - m > 0$;
    - If $m < n$ then $-m > -n$;
    - $n \cdot (-m) = -nm$;
    - $n(m - k) = nm - nk$;
    - If $n < m$ then $n - k < m - k$.
13. Suppose you have an infinite chessboard with a natural number written in each square. The value in each square is the average of the values of the four neighboring squares. Prove that all the values on the chessboard are equal.
14. Prove that every natural number can be written as a sum of *distinct non-consecutive* Fibonacci numbers. For example, $22 = 1 + 3 + 5 + 13$ is not allowed, since 3 and 5 are consecutive Fibonacci numbers, but $22 = 1 + 21$ is allowed.

CHAPTER EIGHTEEN

THE NATURAL NUMBERS AND INDUCTION IN LEAN

The goal of this chapter is to give a more axiomatic, foundational account of the natural numbers and its basic operations. First, we will do this informally, showing how operations like addition and multiplication can be defined using the principle of recursion, and showing how some of their basic properties can be proved using induction. Then we will see how this plays out in the Lean theorem prover, using Lean's built-in mechanisms for induction and recursion.

18.1 Defining Arithmetic Operations

Let $\mathbb{N}$ be the set of natural numbers with least element 0, and let $\mathrm{succ}(m) = m + 1$ be the successor function. The structure $(\mathbb{N}, 0, \mathrm{succ})$ satisfies the following clauses:

- $0 \neq \mathrm{succ}(m)$ for any m in $\mathbb{N}$.
- For every m and n in $\mathbb{N}$, if $m \neq n$, then $\mathrm{succ}(m) \neq \mathrm{succ}(n)$. In other words, succ is *injective*.
- If A is any subset of $\mathbb{N}$ with the property that 0 is in A and whenever n is in A then $\mathrm{succ}(n)$ is in A, then $A = \mathbb{N}$.

The last clause can be reformulated as the principle of induction:

> Suppose $P(n)$ is any property of natural numbers, such that P holds of 0, and for every n, $P(n)$ implies $P(\mathrm{succ}(n))$. Then every P holds of every natural number.

Remember that this principle can be used to justify the principle of definition by recursion:

> Let A be any set, a be any element of A, and let $g(n, m)$ be any function from $\mathbb{N} \times A$ to A. Then there is a unique function $f : \mathbb{N} \to A$ satisfying the following two clauses:
>
> - $f(0) = a$
> - $f(\mathrm{succ}(n)) = g(n, f(n))$ for every n in N.

We can use the principle of recursive definition to define addition with the following two clauses:

$$m + 0 = m$$
$$m + \mathrm{succ}(n) = \mathrm{succ}(m + n)$$

Note that we are fixing m, and viewing this as a function of n. If we write $1 = \mathrm{succ}(0)$, $2 = \mathrm{succ}(1)$, and so on, it is easy to prove $n + 1 = \mathrm{succ}(n)$ from the definition of addition.

We can proceed to define multiplication using the following two clauses:

$$m \cdot 0 = 0$$
$$m \cdot \mathrm{succ}(n) = m \cdot n + m$$

We can also define a predecessor function by

$$\begin{aligned} \mathrm{pred}(0) &= 0 \\ \mathrm{pred}(\mathrm{succ}(n)) &= n, \end{aligned}$$

and "truncated subtraction" by

$$\begin{aligned} m \dot{-} 0 &= 0 \\ m \dot{-} (\mathrm{succ}(n)) &= \mathrm{pred}(m \dot{-} n). \end{aligned}$$

With these definitions and the induction principle, one can prove all the following identities:

- $n \neq 0$ implies $\mathrm{succ}(\mathrm{pred}(n)) = n$
- $0 + n = n$
- $\mathrm{succ}(m) + n = \mathrm{succ}(m + n)$
- $(m + n) + k = m + (n + k)$
- $m + n = n + m$
- $m(n + k) = mn + mk$
- $0 \cdot n = 0$
- $1 \cdot n = x$
- $(mn)k = m(nk)$
- $mn = nm$

We will do the first five here, and leave the remaining ones as exercises.

Proposition. For every natural number n, if $n \neq 0$ then $\mathrm{succ}(\mathrm{pred}(n)) = n$.

Proof. By induction on n. We have ruled out the case where n is 0, so we only need to show that the claim holds for $\mathrm{succ}(n)$. But in that case, we have $\mathrm{succ}(\mathrm{pred}(\mathrm{succ}(n)) = \mathrm{succ}(n)$ by the second defining clause of the predecessor function.

Proposition. For every n, $0 + n = n$.

Proof. By induction on n. We have $0 + 0 = 0$ by the first defining clause for addition. And assuming $0 + n = n$, we have $0 + \mathrm{succ}(n) = \mathrm{succ}(0 + n) = n$, using the second defining clause for addition.

Proposition. For every m and n, $\mathrm{succ}(m) + n = \mathrm{succ}(m + n)$.

Proof. Fix m and use induction on n. Then $n = 0$, we have $\mathrm{succ}(m) + 0 = \mathrm{succ}(m) = \mathrm{succ}(m + 0)$, using the first defining clause for addition. Assuming the claim holds for n, we have

$$\begin{aligned} \mathrm{succ}(m) + \mathrm{succ}(n) &= \mathrm{succ}(\mathrm{succ}(m) + n) \\ &= \mathrm{succ}(\mathrm{succ}(m + n)) \\ &= \mathrm{succ}(m + \mathrm{succ}(n)), \end{aligned}$$

using the inductive hypothesis and the second defining clause for addition.

Proposition. For every m, n, and k, $(m + n) + k = m + (n + k)$.

Proof. By induction on k. The case where $k = 0$ is easy, and in the induction step we have

$$\begin{aligned} (m + n) + \mathrm{succ}(k) &= \mathrm{succ}((m + n) + k) \\ &= \mathrm{succ}(m + (n + k)) \\ &= m + \mathrm{succ}(n + k) \\ &= m + (n + \mathrm{succ}(k))) \end{aligned}$$

using the inductive hypothesis and the definition of addition.

Proposition. For every pair of natural numbers m and n, $m + n = n + m$.

Proof. By induction on n. The base case is easy using the second proposition above. In the inductive step, we have

$$\begin{aligned} m + \mathrm{succ}(n) &= \mathrm{succ}(m + n) \\ &= \mathrm{succ}(n + m) \\ &= \mathrm{succ}(n) + m \end{aligned}$$

using the third proposition above.

18.2 Induction and Recursion in Lean

Internally, in Lean, the natural numbers are defined as a type generated inductively from an axiomatically declared `zero` and `succ` operation:

```
inductive nat : Type
| zero : nat
| succ : nat → nat
```

If you click the button that copies this text into the editor in the online version of this textbook, you will see that we wrap it with the phrases `namespace hide` and `end hide`. This puts the definition into a new "namespace," so that the identifiers that are defined are `hide.nat`, `hide.nat.zero` and `hide.nat.succ`, to avoid conflicting with the one that is in the Lean library. Below, we will do that in a number of places where our examples duplicate objects defined in the library. The unicode symbol ℕ, entered with `\N` or `\nat`, is a synonym for `nat`.

Declaring `nat` as an inductively defined type means that we can define functions by recursion, and prove theorems by induction. For example, these are the first two recursive definitions presented in the last chapter:

```
open nat

def two_pow : ℕ → ℕ
| 0        := 1
| (succ n) := 2 * two_pow n

def fact : ℕ → ℕ
| 0        := 1
| (succ n) := (succ n) * fact n
```

Addition and numerals are defined in such a way that Lean recognizes `succ n` and `n + 1` as essentially the same, so we could instead write these definitions as follows:

```
def two_pow : ℕ → ℕ
| 0       := 1
| (n + 1) := 2 * two_pow n

def fact : ℕ → ℕ
| 0       := 1
| (n + 1) := (n + 1) * fact n
```

If we wanted to define the function `m^n`, we would do that by fixing `m`, and writing doing the recursion on the second argument:

```
def pow : ℕ → ℕ → ℕ
| m 0         := 1
| m (n + 1)   := m * pow m n
```

Lean is also smart enough to interpret more complicated forms of recursion, like this one:

```
def fib : ℕ → ℕ
| 0         := 0
| 1         := 1
| (n + 2)   := fib (n + 1) + fib n
```

In addition to defining functions by recursion, we can prove theorems by induction. In Lean, each clause of a recursive definition results in a new identity. For example, the two clauses in the definition of `pow` above give rise to the following two theorems:

```
theorem pow_zero (n : ℕ) : pow n 0 = 1 := rfl
theorem pow_succ (m n : ℕ) : pow m (n+1) = m * pow m n := rfl
```

Notice that we could alternatively have used `(pow m n) * m` in the second clause of the definition of `pow`. Of course, we can prove that the two definitions are equivalent using the commutativity of multiplication, but, using a proof by induction, we can also prove it using only the associativity of multiplication, and the properties `1 * m = m` and `m * 1 = m`. This is useful, because the power function is also often used in situations where multiplication is not commutative, such as with matrix multiplication. The theorem can be proved in Lean as follows:

```
theorem pow_succ' (m n : ℕ) : pow m (succ n) = (pow m n) * m :=
nat.rec_on n
  (show pow m (succ 0) = pow m 0 * m, from calc
    pow m (succ 0) = m * pow m 0 : by rw pow_succ
               ... = m * 1       : by rw pow_zero
               ... = m           : by rw mul_one
               ... = 1 * m       : by rw one_mul
               ... = pow m 0 * m : by rw pow_zero)
  (assume n,
    assume ih : pow m (succ n) = pow m n * m,
    show pow m (succ (succ n)) = pow m (succ n) * m, from calc
      pow m (succ (succ n)) = m * (pow m (succ n)) : by rw pow_succ
                        ... = m * (pow m n * m)    : by rw ih
                        ... = (m * pow m n) * m    : by rw mul_assoc
                        ... = pow m (succ n) * m   : by rw pow_succ)
```

This is a typical proof by induction in Lean. It begins with the phrase `nat.induction_on n`, and is followed by the base case and the inductive hypothesis. The proof can be shortened using `rewrite` and `simp`:

```
theorem pow_succ' (m n : ℕ) : pow m (succ n) = (pow m n) * m :=
nat.rec_on n
  (show pow m (succ 0) = pow m 0 * m,
    by rw [pow_succ, pow_zero, mul_one, one_mul])
  (assume n,
    assume ih : pow m (succ n) = pow m n * m,
    show pow m (succ (succ n)) = pow m (succ n) * m,
      by simp [pow_succ, ih])
```

Remember that you can write a `rewrite` proof incrementally, checking the error messages to make sure things are working so far, and to see how far Lean got.

In any case, the power function is already defined in the Lean library as `pow_nat`. (It is defined generically for any type that has a multiplication; the `nat` in `pow_nat` refers to the fact that the exponent is a natural number.) The definition is essentially the one above, and the theorems above are also there:

```
import algebra.group_power
local infix ` ^ ` := pow_nat

#check @pow_nat
#check @pow_zero
#check @pow_succ
#check @pow_succ'

variables m n : ℕ

#check m^n
```

As another example of a proof by induction, here is a proof of the identity `m^(n + k) = m^n * m^k`.

```
theorem pow_add (m n k : ℕ) : m^(n + k) = m^n * m^k :=
nat.rec_on k
  (show m^(n + 0) = m^n * m^0, from calc
    m^(n + 0) = m^n       : by rw add_zero
          ... = m^n * 1   : by rw mul_one
          ... = m^n * m^0 : by rw pow_zero)
  (assume k,
    assume ih : m^(n + k) = m^n * m^k,
    show m^(n + succ k) = m^n * m^(succ k), from calc
      m^(n + succ k) = m^(succ (n + k)) : by rw nat.add_succ
                 ... = m^(n + k) * m    : by rw pow_succ'
                 ... = m^n * m^k * m    : by rw ih
                 ... = m^n * (m^k * m)  : by rw mul_assoc
                 ... = m^n * m^(succ k) : by rw pow_succ')
```

Notice the same pattern. This time, we do induction on `k`, and the base case and inductive step are routine. Once again, with a bit of cleverness, we can shorten the proof with `rewrite`:

```
theorem pow_add (m n k : ℕ) : m^(n + k) = m^n * m^k :=
nat.rec_on k
  (show m^(n + 0) = m^n * m^0,
    by rewrite [add_zero, pow_zero, mul_one])
  (assume k,
    assume ih : m^(n + k) = m^n * m^k,
    show m^(n + succ k) = m^n * m^(succ k),
     by rewrite [nat.add_succ, pow_succ', ih, mul_assoc, pow_succ'])
```

You should not hesitate to use `calc`, however, to make the proofs more explicit. Remember that you can also use `calc` and `rewrite` together, using `calc` to structure the calculational proof, and using `rewrite` to fill in each justification step.

18.3 Defining the Arithmetic Operations in Lean

In fact, addition and multiplication are defined in Lean essentially as described in Section 18.1. The defining equations for addition hold by reflexivity, but they are also named `add_zero` and `add_succ`:

```
open nat

variables m n : ℕ
```

```
example : m + 0 = m := add_zero m
example : m + succ n = succ (m + n) := add_succ m n
```

Similarly, we have the defining equations for the predecessor function and multiplication:

```
open nat

#check @pred_zero
#check @pred_succ
#check @mul_zero
#check @mul_succ
```

Here are the five propositions proved in Section 18.1.

```
theorem succ_pred (n : ℕ) : n ≠ 0 → succ (pred n) = n :=
nat.rec_on n
  (assume H : 0 ≠ 0,
    show succ (pred 0) = 0, from absurd rfl H)
  (assume n,
    assume ih,
    assume H : succ n ≠ 0,
    show succ (pred (succ n)) = succ n,
      by rewrite pred_succ)

theorem zero_add (n : nat) : 0 + n = n :=
nat.rec_on n
  (show 0 + 0 = 0, from rfl)
  (assume n,
    assume ih : 0 + n = n,
    show 0 + succ n = succ n, from
      calc
    0 + succ n = succ (0 + n) : rfl
      ... = succ n : by rw ih)

theorem succ_add (m n : nat) : succ m + n = succ (m + n) :=
nat.rec_on n
  (show succ m + 0 = succ (m + 0), from rfl)
  (assume n,
    assume ih : succ m + n = succ (m + n),
    show succ m + succ n = succ (m + succ n), from
      calc
    succ m + succ n = succ (succ m + n) : rfl
      ... = succ (succ (m + n)) : by rw ih
      ... = succ (m + succ n) : rfl)

theorem add_assoc (m n k : nat) : m + n + k = m + (n + k) :=
nat.rec_on k
  (show m + n + 0 = m + (n + 0), by rw [add_zero, add_zero])
  (assume k,
    assume ih : m + n + k = m + (n + k),
    show m + n + succ k = m + (n + (succ k)), from calc
      m + n + succ k = succ (m + n + k)   : by rw add_succ
                 ... = succ (m + (n + k)) : by rw ih
                 ... = m + (n + succ k)   : by rw add_succ)

theorem add_comm (m n : nat) : m + n = n + m :=
nat.rec_on n
```

```
  (show m + 0 = 0 + m, by rewrite [add_zero, zero_add])
  (assume n,
    assume ih : m + n = n + m,
    show m + succ n = succ n + m, from calc
      m + succ n = succ (m + n) : by rw add_succ
             ... = succ (n + m) : by rw ih
             ... = succ n + m   : by rw succ_add)
```

18.4 Exercises

1. Give an informal but detailed proof that for every natural number n, $1 \cdot n = n$.
2. Prove the multiplication is associative and commutative, in the same way.
3. Prove that multiplication distributes over addition: for every natural numbers m, n, and k, $m(n+k) = mn + mk$.
4. Prove $(m^n)^k = m^{nk}$.
5. Formalize all these theorems in Lean.