Logic and Proof

Jeremy Avigad
Robert Y. Lewis
Floris van Doorn

Version 81cd57a, updated at 2015-09-03 17:16:28 -0400

Contents

1

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 $a = \sqrt{2}b$, 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. By the inductive hypothesis, each of m 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 judgement 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 follows: 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:

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

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 reasoning, 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 Yourself." 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.rat
open eq.ops nat

theorem sqrt_two_irrational {a b : ℕ} (co : coprime a b) : a^2 ≠ 2 * b^2 :=
assume H : a^2 = 2 * b^2,
```

```
have even (a^2), from even_of_exists (exists.intro _ H),
have even a, from even_of_even_pow this,
obtain c (aeq : a = 2 * c), from exists_of_even this,
have 2 * (2 * c^2) = 2 * b^2, by rewrite [-H, aeq, *pow_two, mul.assoc, mul.left_comm c],
have 2 * c^2 = b^2, from eq_of_mul_eq_mul_left dec_trivial this,
have even (b^2), from even_of_exists (exists.intro _ (eq.symm this)),
have even b, from even_of_even_pow this,
have 2 ∣ gcd a b, from dvd_gcd (dvd_of_even `even a`) (dvd_of_even `even b`),
have 2 ∣ 1, from co ▸ this,
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 characterize 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 specification language.

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 ineractive 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 These Notes

Lean is a new theorem prover, and is still under development. Similarly, these notes are being written on the fly as the class proceeds, and parts will be sketchy, buggy, and incomplete. They will therefore at best serve as a supplement to class notes and the textbook, Daniel Velleman's *How to Prove it: A Structured Approach*. Please bear with us! Your feedback will be quite helpful to us.

2

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 "extra logical" 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 rules governing the terms "and," "or," "not," and "if … then." Following the model described in the introduction, each such construction will be analyzed in three ways:

- with examples of the way it is used and employed in informal (mathematical) arguments
- with a formal, symbolic representation
- with an implementation in *Lean*

2.3 Rules of Inference

Implication

The first pattern of reasoning 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 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. In a system of natural deduction, it is expressed as follows:

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

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.

In Lean, the inference is expressed as follows:

```
variables (A B : Prop)
premises (H₁ : A → B) (H₂ : A)

example : B :=
show B, from H₁ H₂
```

The first command declares two variables, `A` and `B`, ranging over propositions. The second line introduces two premises, namely, `A → B` and `A`. The next line asserts, as an example, that `B` follows from the premises. The proof is written simply $\texttt{H}_1\ \texttt{H}_2$: think of this as the premise $\texttt{H}_1$ "applied to" the premise $\texttt{H}_2$.

You can enter the arrow by writing `\to` or `\imp` or `\r`. You can enter $\texttt{H}_1$ by typing `H\_1`. You can use any reasonable alphanumeric identifier for a hypothesis; the letter "H" is a conventional choice. The identifier `H1` is a different from $\texttt{H}_1$, but you can also use that, if you prefer.

The rule for proving 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.

$$\dfrac{\dfrac{}{H : A}\\ \vdots \\ B}{A \to B}\ \to\text{I}, H$$

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

In Lean, this inference takes the following form:

```
variables (A B : Prop)

example : A → B :=
assume H : A,
show B, from sorry
```

To prove `A → B`, we assume `A`, with label `H`, and show `B`. Here, the word `sorry` indicates that the proof is omitted. In this case, this is necessary; since `A` and `B` are arbitrary propositions,

there is no way to prove `B` from `A`. In general, though, `A` and `B` will be compound expressions, and you are free to use the hypothesis `H : A` to prove `B`.

Using `sorry`, we can illustrate the implication elimination rule alternatively as follows:

```
variables (A B : Prop)

example : B :=
have H₁ : A → B, from sorry,
have H₂ : A, from sorry,
show B, from H₁ H₂
```

We will adopt this convention below, using `sorry` to stand for parts of a proof that could be spelled out, when the variables involved are replaced by more complex assertions.

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 natural deduction, the rule reads as follows:

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

In Lean, the rule is denoted `and.intro`:

```
variables (A B : Prop)

example : A ∧ B :=
have H₁ : A, from sorry,
have H₂ : B, from sorry,
show A ∧ B, from and.intro H₁ H₂
```

You can enter the wedge symbol by typing `\and`.

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 natural deduction, these patterns are rendered as follows:

$$\frac{A \wedge B}{A}\ \wedge\mathrm{E}_1 \qquad \frac{A \wedge B}{B}\ \wedge\mathrm{E}_2$$

In Lean, the inferences are known as `and.left` and `and.right`:

```
variables (A B : Prop)

example : A :=
have H : A ∧ B, from sorry,
show A, from and.left H

example : B :=
have H : A ∧ B, from sorry,
show B, from and.right H
```

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 natural deduction, the rule reads as follows:

$$\dfrac{\dfrac{\overline{A}}{\vdots}}{\dfrac{\bot}{\neg A}\,\neg\text{I}}$$

In Lean, it is illustrated by the following:

```
variable A : Prop

example : ¬ A :=
assume H : A,
show false, from sorry
```

You can enter the negation symbol by typing `\not`.

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:

$$\dfrac{\neg A \qquad A}{\bot}\,\neg\text{E}$$

And in Lean, it is implemented in the following way:

```
variable A : Prop

example : false :=
have H₁ : ¬ A, from sorry,
have H₂ : A, from sorry,
show false, from H₁ H₂
```

Notice that the negation elimination rule is expressed in a manner similar to implication elimination: the label asserting the negation comes first, and by "applying" the proof of the negation to the proof of the positive fact, we obtain a proof of falsity.

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

What are the rules governing $\bot$? In natural deduction, 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, natural deduction provides a rule that allows us to conclude anything from a contradiction:

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

The elimination rule also has the fancy Latin name, *ex falso sequitur quodlibet*, which means "anything you want follows from falsity." In Lean it is implemented as follows:

```
variable A : Prop

example : A :=
have H : false, from sorry,
show A, from false.elim H
```

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. In Lean, we can use `true.intro` for this rule, or the abbreviation `trivial`.

```
example : true :=
show true, by trivial
```

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 terms of natural deduction, the two introduction rules are as follows:

$$\frac{A}{A \vee B} \vee \mathrm{I_l} \qquad \frac{B}{A \vee B} \vee \mathrm{I_r}$$

Here, the l and r stand for "left" and "right". In Lean, they are implemented as follows:

```
variables (A B : Prop)

example : A ∨ B :=
have H : A, from sorry,
show A ∨ B, from or.inl H

example : A ∨ B :=
have H : B, from sorry,
show A ∨ B, from or.inr H
```

You can enter the vee symbol by typing `\or`. The identifiers `inl` and `inr` stand for "insert left" and "insert right," respectively.

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 natural deduction, this pattern is expressed as follows:

$$\dfrac{A \vee B \qquad \begin{array}{c}\overline{A}\\ \vdots \\ C\end{array} \qquad \begin{array}{c}\overline{B}\\ \vdots \\ C\end{array}}{C}\ \vee\text{E}$$

And here it is in Lean:

```
variables (A B C : Prop)

example : 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)
```

What makes this pattern confusing is that it requires two instances of nested hypothetial 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.

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:

$$\dfrac{\begin{array}{cc}\overline{A} & \overline{B} \\ \vdots & \vdots \\ B & A\end{array}}{A \leftrightarrow B}\ \leftrightarrow \mathrm{I}$$

And here is in Lean:

```
variables (A B : Prop)

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

You enter the symbol ↔ by typing `\iff` or `\lr` (for the left-right arrow). Notice that you can re-use the letter `H` for the hypothesis, since the two branches of the proof are independent.

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:

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

Lean defines the rules `iff.and_elim_left` and `iff.and_elim_right`, but also provides the abbreviations `iff.mp` (for "modus ponens") and `iff.mpr` (for modus ponens reverse).

```
variables (A B : Prop)

example : B :=
have H₁ : A ↔ B, from sorry,
have H₂ : A, from sorry,
show B, from iff.mp H₁ H₂

example : A :=
have H₁ : A ↔ B, from sorry,
have H₂ : B, from sorry,
show A, from iff.mpr H₁ H₂
```

Proof by Contradiction

We saw an example of an informal argument that implictly 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:

$$\dfrac{\dfrac{}{\neg A}}{\begin{array}{c}\vdots\\ \bot\end{array}}{A}$$

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)
```

2.4 Writing Proofs in Natural Deduction

As noted in Chapter Introduction, there are two common styles for writing natural deduction derivations. (The word "derivation" is often used to connote a formal proof instead of an informal one. When talking about natural deduction, we will use the words "derivation"

and "proof" interchangeably.) In both cases, proofs are presented on paper as trees, with the conclusion at the theorem at the root, and hypotheses up at the leaves. In the first style of presentation, the set of hypotheses is written explicitly at every node of the tree. This is helpful because some rules (namely, implication introduction, negation introduction, or elimination, and proof by contradiction) change the set of hypotheses, by canceling a local or temporary assumption. Nonetheless, we will use a style of presentation that leaves this information implicit, so that each node of the tree is labelled with an explicit formula. Some people like to label each inference with the rule that is used, but that is usually clear from the context, so we will omit that as well. But when a rule cancels a hypothesis, we will make that clear in the following way: we will label all instances of the hypothesis at the leaves with a letter, like "x," and then we will use that letter to annotate the place where the rule is canceled.

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

In addition to the rules listed in the last section, there is one additional rule that is central to the system, namely the assumption rule. It works like this: at any point, you can assume a hypothesis, A. The way to read such a one-line proof is this: assuming A, we have proved A. Without this rule, there would be no way of getting a proof off the ground! After all, every rule listed in the last section has premises, which is to say, it can only be applied to derivations that have been constructed previously.

Let us consider a few examples. 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 the uncanceled hypotheses. Here is a proof of $A \wedge (B \vee C) \to (A \wedge B) \vee (A \wedge C)$:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{y : A \wedge (B \vee C)}}{B \vee C} \qquad \dfrac{\dfrac{\dfrac{\overline{y : A \wedge (B \vee C)}}{A} \qquad \overline{x : B}}{A \wedge B}}{(A \wedge B) \vee (A \wedge C)} \qquad \dfrac{\dfrac{\dfrac{\overline{y : A \wedge (B \vee C)}}{A} \qquad \overline{x : C}}{A \wedge C}}{(A \wedge B) \vee (A \wedge C)}}{(A \wedge B) \vee (A \wedge C)}\, x}{(A \wedge (B \vee C)) \to ((A \wedge B) \vee (A \wedge C))}\, y$$

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

1. First, work 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 and try to prove any open goals by splitting on cases, considering A in one case and B in the other.

3. If all else fails, use a proof by contradiction.

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

$$\dfrac{\dfrac{\dfrac{\dfrac{}{A \to (B \to C)}\,y \qquad \dfrac{\dfrac{}{A \wedge B}\,x}{A}}{B \to C} \qquad \dfrac{\dfrac{}{A \wedge B}\,x}{B}}{C}}{\dfrac{C}{A \wedge B \to C}\,x}\,y$$
$$(A \to (B \to C)) \to (A \wedge B \to C)$$

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.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\neg(A \vee B)}\,z \qquad \dfrac{\dfrac{}{A}\,x}{A \vee B}}{\bot}}{\neg A}\,x \qquad \dfrac{\dfrac{\dfrac{}{\neg(A \vee B)}\,z \qquad \dfrac{\dfrac{}{B}\,y}{A \vee B}}{\bot}}{\neg B}\,y}{\neg A \wedge \neg B}$$
$$\neg(A \vee B) \to \neg A \wedge \neg B\;z$$

2.5 Writing Proofs in Lean

We will see that Lean has mechanisms for modeling proofs at a higher level than natural deduction derivations. At the same time, you can also carry out low-level inferences, and carry out proofs that mirror natural deduction proofs quite closely. Here is a Lean representation of the first example in the previous section:

```
variables (A B C : Prop)

example : 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)
```

The first line declares propositional variables `A`, `B`, and `C`. The line that begins with the keyword `example` declares the theorem to be proved, and the notation `:=` indicates that the proof will follow. The line breaks and indentation are only for the purposes of readability; Lean would do just was well if the entire proof were written as one run-on line.

Here are some additional notes:

- It is often important to name a theorem for future proof. Lean allows us to do that, using one of the keywords `theorem`, `lemma`, `proposition`, `corollary`, followed by the name of the proof.
- You can omit a label in a `have` statement. You can then refer to that fact using the label `this`, until the next anonymoyus `have`. Alternatively, at any point later in the proof, you can refer to the fact by putting the assertion between backticks.
- One can also omit the label in an `assumption` by using the keyword `suppose` instead.
- Rather than declare variables beforehand, you can declare them in parentheses before the colon the marks the statemet of the theorem.

With these features, the previous proof can be written as follows:

```
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`
    (suppose B,
      have A ∧ B, from and.intro `A` `B`,
      show (A ∧ B) ∨ (A ∧ C), from or.inl this)
    (suppose C,
      have A ∧ C, from and.intro `A` `C`,
      show (A ∧ B) ∨ (A ∧ C), from or.inr this)
```

In fact, such a presentation provides Lean with more information than is really necessary to construct an axiomatic proof. The word `assume` can be replaced by the symbol λ, assertions can be omitted from an `assume` when they can be inferred from context, the

justification of a have statement can be inserted in places where the label was otherwise used, and one can omit the `show` clauses, giving only the justification. As a result, the previous proof can be written in an extremely abbreviated form:

```
example (A B C : Prop) : A ∧ (B ∨ C) → (A ∧ B) ∨ (A ∧ C) :=
λ H₁, or.elim (and.right H₁)
  (λ H₄, or.inl (and.intro (and.left H₁) H₄))
  (λ H₄, or.inr (and.intro (and.left H₁) H₄))
```

Such proofs tend to be hard to write, read, understand, maintain, and debug. In this text, we will favor structure and readability over brevity.

The next proof in the previous section can be rendered in Lean as follows:

```
variables (A B C : Prop)

example : (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)
```

And the last proof can be rendered as follows:

```
variables (A B : Prop)

example : ¬ (A ∨ B) → ¬ A ∧ ¬ B :=
assume H : ¬ (A ∨ B),
have ¬ A, from
  suppose A,
  have A ∨ B, from or.inl `A`,
  show false, from H this,
have ¬ B, from
  suppose B,
  have A ∨ B, from or.inr `B`,
  show false, from H this,
show ¬ A ∧ ¬ B, from and.intro `¬ A` `¬ B`
```

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 :=
suppose A,          -- assume the antecedent
show A, from this   -- use the assuption to establish the conclusion
```

Notice that you can use `sorry` as a temporary placeholder while writing proofs.

```
example (A B : Prop) : A ∧ B → B ∧ A :=
assume H : A ∧ B,
have H1 : A, from and.left H,
have H2 : B, from and.right H,
show B ∧ A, from sorry
```

This enables you to check the proof to make sure it is correct modulo the `sorry`, before you go on to replace the `sorry` with an actual proof.

Here is another useful trick: try replacing the `sorry` by an underscore character, `_`. This asks the Lean parser to guess what should go there, based on the context. In this case, Lean does not succeed, and gives you error message when you try to check the proof. But the error message is informative: it tells you what you need to prove, and what is available in the context for you to use.

2.6 Writing Informal Proofs

Remember that one goal of this course is to teach you to write ordinary (mathematical) proofs as well formal proofs in natural deduction and formally verified proofs in Lean. The fact that natural deduction and Lean's proof language are designed to model some aspects of informal proof does *not* mean that your informal proofs should look like natural deduction derivations or proofs in Lean! There are important differences between formal languages and informal language that you should keep in mind.

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.

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.

Perhaps the best way 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.

2.7 Theorems and Derived Rules

In the examples above, we showed that, given $A \vee B$ and $\neg A$, we can derive B in natural deduction. This is a common pattern of inference, and, having justified it once, you might reasonably want to use it freely as a new one-step inference. Similarly, having proved $A \to B$ equivalent to $\neg A \vee B$, or $\neg(A \vee B)$ equivalent to $\neg A \wedge \neg B$, one might feel justified in replacing one by the other in any expression.

Indeed, this is how informal mathematics works: we start with basic patterns of inference, but over time we learn to recognize more complex patterns, and begin to apply them freely in our proofs. A single step in the informal argument in the solution to "Malice and Alice," or any mathematical proof, usually requires many more steps to spell out in a formal calculus. Moreover, in ordinary mathematics, one we prove a proposition or theorem, we can freely invoke it in another proof later on.

One can extend natural deduction with various mechanisms to abbreviate such "derived rules." We will not do so here, however. Natural deduction is designed to model the low-level mechanics of a proof and let us reason about deduction "from the outside"; we will not use it to write long proofs.

In formal verification, however, the goal is to build complex proofs, developing libraries for formalized mathematics along the way. To that end, Lean allows you to name the theorems you prove:

```
theorem not_and_not_of_not_or (A B : Prop) :
  ¬ (A ∨ B) → ¬ A ∧ ¬ B :=
assume H : ¬ (A ∨ B),
have ¬ A, from
  suppose A,
  have A ∨ B, from or.inl `A`,
  show false, from H this,
have ¬ B, from
  suppose B,
  have A ∨ B, from or.inr `B`,
  show false, from H this,
show ¬ A ∧ ¬ B, from and.intro `¬ A` `¬ B`
```

Here we follow the convention of describing the conclusion of the theorem first (`not_and_not`), followed by the hypotheses (in this case, `not_or`), separated by `of`. Thereafter, we can use the theorem as a new rule of inference:

```
variables (C D : Prop)

example : ¬ (C ∨ D) → ¬ C ∧ ¬ D :=
assume H : ¬ (C ∨ D),
show ¬ C ∧ ¬ D, from not_and_not_of_not_or C D H
```

Notice that `not_and_not_of_not_or` takes, as arguments, the two propositions `C` and `D` to which we want to instantiate the theorem, followed by the hypothesis `H`.

We can tell Lean to make the first two arguments implicit, by changing `(A B : Prop)` to `{A B : Prop}`. The curly braces ask Lean to infer the values of these arguments from the context. With that change, we can write the preceding proof as follows:

```
variables (C D : Prop)

example : ¬ (C ∨ D) → ¬ C ∧ ¬ D :=
assume H : ¬ (C ∨ D),
show ¬ C ∧ ¬ D, from not_and_not_of_not_or H
```

Here is a more interesting example: first we show, independently, that each of `¬ A` and `¬ B` follows from `¬ (A ∨ B)`, and then we use these facts to prove `not_and_not_of_not_or`.

```
variables {A B : Prop}

theorem not_of_not_or_left : ¬ (A ∨ B) → ¬ A :=
assume H : ¬ (A ∨ B),
show ¬ A, from
  suppose A,
  have A ∨ B, from or.inl `A`,
  show false, from H this

theorem not_of_not_or_right : ¬ (A ∨ B) → ¬ B :=
assume H : ¬ (A ∨ B),
show ¬ B, from
  suppose B,
  have A ∨ B, from or.inr `B`,
  show false, from H this

theorem not_and_not_of_not_or : ¬ (A ∨ B) → ¬ A ∧ ¬ B :=
assume H : ¬ (A ∨ B),
have ¬ A, from not_of_not_or_left H,
have ¬ B, from not_of_not_or_right H,
show ¬ A ∧ ¬ B, from and.intro `¬ A` `¬ B`
```

Later, we will see that Lean has an expansive library of theorems. Eventually, Lean will also have automation that will fill in small steps automatically. In elementary exercises, however, we will expect you to carry out such proofs by hand.

To summarize our expectations in this course:

- When we ask you to prove something in natural deduction, our goal is to make you work with the precise, formal rules of the system, so you should not appeal to external rules unless we explicitly say you can.
- In interactive theorem proving, the main goal is to have the computer certify the proof as correct, and in that respect, automation and facts from the library are fair game. To learn to use the system, however, it is helpful to prove elementary theorems by hand. In this class, we will try to be explicit about what we would like you to use in the exercises we assign.
- When writing informal proofs, it is a judgment call as to what prior patterns of reasoning and background facts you may appeal to. In a classroom setting, the goal may be to demonstrate mastery of the subject to the instructors, in which case, context should dictate what is allowable (and it is always a good idea to err on the side of caution). In real life, your goal is to convince your target audience, and you will have to rely on convention and experience to judge what patterns of inference you can put forth, and how much detail you need to use.

2.8 Classical Reasoning

In informal mathematics, it is usually clearer to give a "direct" proof of a theorem, rather than using proof by contradiction. But proof by contradiction is sometimes necessary, and, at a foundational level, it can be used to derive other classical patterns of reasoning.

For example, we have seen that if you know $A \vee B$, you can use that knowledge to reason on cases, assuming first A, and then B. In mathematical arguments, however, 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:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\neg(A \vee \neg A)}\,y \qquad \dfrac{\dfrac{}{A}\,x}{A \vee \neg A}}{\bot}}{\neg A}\,x}{A \vee \neg A} \qquad \dfrac{}{\neg(A \vee \neg A)}\,x$$

$$\dfrac{\bot}{A \vee \neg A}\,y$$

Here is the same proof rendered in Lean:

```
open classical

variable (A : Prop)
```

```
example : A ∨ ¬ A :=
by_contradiction
  (assume H : ¬ (A ∨ ¬ A),
    have ¬ A, from
      suppose A,
      have A ∨ ¬ A, from or.inl this,
      show false, from H this,
    have A ∨ ¬ A, from or.inr `¬ A`,
    show false, from H this)
```

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)
  (suppose A, or.inl this)
  (suppose ¬ A, or.inr this)
```

Or even more simply:

```
open classical

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

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.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\neg(A \wedge \neg B)}\,z \qquad \dfrac{\dfrac{}{A}\,y \qquad \dfrac{}{\neg B}\,x}{A \wedge \neg B}}{\bot}}{B}\,x}{A \to B}\,y}{\neg(A \wedge \neg B) \to (A \to B)}\,z$$

Here is the same proof, rendered in Lean:

```
open classical

variables (A B : Prop)

example (H : ¬ (A ∧ ¬ B)) : A → B :=
```

```
suppose A,
show B, from
  by_contradiction
    (suppose ¬ B,
      have A ∧ ¬ B, from and.intro `A` this,
      show false, from H this)
```

2.9 Some Logical Identities

For reference, the following is a list of commonly used propositional equivalences.

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 them can be derived in natural deduction, and in Lean, using the rules and patterns of inference discussed in this Chapter.

3

Truth Tables and Semantics

In the last chapter, we saw how to prove a formula of propositional logic from hypotheses. Some formulas are provable outright, from no hypotheses at all, such as the formula $A \to A$.

It seems intuitively clear that, in contrast, we cannot prove the formula $A \to B$ without additional assumptions. Try it in Lean:

```
variables A B : Prop

example : A → B :=
assume H : A,
show B, from sorry
```

It should seem unlikely that there is an argument we could put in place of the "sorry" to complete this proof. After all, `B` could be false!

What do we mean by "false," exactly? We used the identifiers `true` and `false` to denote certain propositions in Lean, but here we are using the words in a difference sense, as values, or judgments, we can assign to propositional formulas. Such evaluations belong 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 drive A from Γ?
- What formulas can be derived from Γ?
- What hypotheses are need to derive A?

The questions we consider semantically are different:

- Given an assignment of truth values to the propositional variables occuring 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?

These syntactic and semantic notions complement each other, in ways we will describe below. But first, we will discuss methods we can use to answer semantic questions like the ones above.

3.1 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 **T** and **F** 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 Classical Reasoning. 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 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. For example, the function v defined by

- $v(A) := \mathbf{T}$
- $v(B) := \mathbf{F}$
- $v(C) := \mathbf{F}$
- $v(D) := \mathbf{T}$

is a truth assignment for the set of variables $\{A, B, C, D\}$.

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 **T** to P and R, and the value **F** to Q.

3.2 Evaluating Formulas

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.
- $\bar{v}(\neg\varphi) = \mathbf{T}$ if $\bar{v}(\varphi)$ is **F**, and vice versa.
- $\bar{v}(\varphi \wedge \psi) = \mathbf{T}$ if $\bar{v}(\varphi)$ and $\bar{v}(\psi)$ are both **T**, and **F** otherwise.
- $\bar{v}(\varphi \vee \psi) = \mathbf{T}$ if at least one of $\bar{v}(\varphi)$ and $\bar{v}(\psi)$ is **T**; otherwise **F**.
- $\bar{v}(\varphi \to \psi) = \mathbf{T}$ if either $\bar{v}(\psi)$ is **T** or $\bar{v}(\varphi)$ is **F**, and **F** otherwise. (Equivalently, $\bar{v}(\varphi \to \psi) = \mathbf{F}$ if $\bar{v}(\varphi)$ is **T** and $\bar{v}(\psi)$ is **F**, and **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
premise 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:

$$\dfrac{\dfrac{\neg A \qquad \overline{H : A}}{\bot}}{A \to B}\, H$$

In Lean:

```
variables A B : Prop
premise 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 $\neg(A \to B)$:

$$\dfrac{\neg B \qquad \dfrac{\overline{H : A \to B} \qquad A}{B}}{\dfrac{\bot}{\neg(A \to B)}}\, H$$

Once again, in Lean:

```
variables A B : Prop
premise HA : A
premise HnB : ¬B

example : ¬ (A → B) :=
assume H : A → B,
have HB : B, from H HA,
show false, from HnB HB
```

3.3 Finding truth assignments

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 occuring 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, by changing the true/false values as you wish.
definition A : Prop := true
definition B : Prop := false
definition C : Prop := true
definition D : Prop := true
definition E : Prop := false

-- Ignore this line.
attribute A B C D E [reducible]

eval is_true ((A ∧ B) ∨ C)
eval is_true (A → D)
eval is_true (C → (D ∨ ¬E))
eval is_true (¬(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.

Truth tables

The second and third semantic questions we asked are a little trickier than the first. Instead of considering one particular truth assignment, they 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 tabel 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*.

We can use Lean to check if whether we have evaluated a formula correctly:

```
/-                  Put your formula here  -/
/-                  \/                     -/
eval let e :=
  λ A     B,        A ∧ (B → A)   in is_true (
( e true  true   ↔  true          ) ∧
( e true  false  ↔  true          ) ∧
( e false true   ↔  false         ) ∧
( e false false  ↔  false         ) )
```

You can replace the formula `A ∧ (B → A)` with any other formula involving the variables `A` and `B`. Then, leaving the first two columns alone, modify the third column by entering the value `true` or `false` corresponding to the assignment in the first two columns. The resulting expression will evaluate to true if and only if you have entered the correct truth values.

(The precise mechanism by which this works is not important right now, but in case you are curious, the idea is as follows. In the expression, the `e` is "locally" defined to be

the function which takes two truth values `A` and `B` as input, and evaluates `A ∧ (B → A)` relative to these inputs. For each line in the truth table, the expression checks whether the formula evaluates to the value you entered, and takes the conjunction of the results.)

Here is the analogous setup for three variables:

```
eval let e :=
  λ A     B     C,       A ∧ (B → C)   in is_true (
( e true  true  true  ↔  true          ) ∧
( e true  true  false ↔  false         ) ∧
( e true  false true  ↔  true          ) ∧
( e true  false false ↔  true          ) ∧
( e false true  true  ↔  false         ) ∧
( e false true  false ↔  false         ) ∧
( e false false true  ↔  false         ) ∧
( e false false false ↔  false         ) )
```

3.4 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.

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

course: 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. 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.

4

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 Propositional Logic. 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.

4.1 Functions, Relations, and Predicates

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.
- For any three natural numbers x, y, and z, is 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, 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 $mul(x, y)$ and $square(x)$. so *mul* takes two arguments, and *square* takes only one.
- Predicates and relations can also be understood in these terms. The predicates $even(x)$ and $prime(x)$ take one argument, while the binary relations $divides(x, y)$ and $lt(x, y)$ 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 $x \cdot y$ intead of $mul(x, y)$, and we write $x < y$ instead of $lt(x, y)$. 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.

What makes the language of first-order logic powerful is that one can 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:

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

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:

$$even(x + y + z), \quad prime((x + 1) \times y \times y), \quad \Box(x + y \times z) = w, \quad x + y < z$$

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

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

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.

One can also declare function and relation symbols in Lean. For example, the symbols we have just discussed could be introduced as follows:

```
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
```

You can enter ℕ with `\nat` or `\N`. In Lean, the `check` command can be used to make sure an expression is well-formed, and determine what kind of expression it is:

```
variables w x y z : ℕ

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

We can even declare infix notation of binary operations and relations:

```
infix + := add
infix * := mul
infix < := lt
```

(Getting notation for numerals `1`, `2`, `3`, … is trickier.) With all this in place, the examples above can be rendered as follows:

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

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.

```
import data.nat
open nat

constant square : ℕ → ℕ
constant prime : 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 course, we will often proceed in this way, telling you explicitly what facts from the library you should use for exercises.

Here are some things to note about the syntax of expression in Lean:

- 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 takes 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 takes 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.

4.2 Quantifiers

There are two more ingredients to the language of first-order logic, namely, the universal and existential 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 in question. Using the universal quantifier, the examples with which we began this previous section can be expressed as follows:

- $\forall x\ ((even(x) \vee odd(x)) \wedge \neg(even(x) \wedge \neg odd(x)))$.
- $\forall x\ (even(x) \to even(x^2))$
- $\forall x\ (even(x) \leftrightarrow 2 \mid x)$
- $\forall x\ \forall y\ \forall z\ (x \mid y \wedge y \mid z \to x \mid z)$.

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.

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

```
import data.nat
open nat

variables x y z : ℕ
```

```
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 y z, x ∣ y → y ∣ z → x ∣ z
```

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.
- In contrast, Lean expects a comma after that universal quantifer, and gives the 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.
- After the quantifier $\forall x$, the variable x is *bound*. For example, the expression $\forall x\ (even(x) \vee odd(x))$ 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 $\forall x\ (even(x) \vee odd(x))$ as being the same as the expression $\forall y\ (even(y) \vee odd(y))$. Lean treats these expressions as the same as well.
- 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. In other words, 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. A couple of chapters from now, we will make the notion of an "intended interpetation" precise, and talk about 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.

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." We will discuss the existential quantifier and its use in a later chapter.

Indeed, to complete the presentation of first-order logic, we need to present the rules of the universal quantifier, the existential quantifier, and equality in natural deduction, and in Lean. In this chapter, we will start with the rules for the universal quantifier, and provide examples of the kinds of mathematical arguments they are intended to model.

4.3 Rules for the Universal Quantifier

In the Introduction 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 woud 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, but showing that it holds of "arbitrary" values a and b. In natural deduction, the proof would look something like this:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{H : a^2 = 2 \times b^2}}{\vdots}}{\bot}}{\neg(a^2 = 2 \times b^2)}\,H}{\forall b\ \neg(a^2 = 2 \times b^2)}}{\forall a\ \forall b\ \neg(a^2 = 2 \times b^2)}$$

Notice that after the hypothesis H 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:

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

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 $\forall x\ A(x)$ as being the same as $\forall y\ A(y)$.

Notice that when we work in first-order logic, we assume that the universal quantifier ranges over some domain. In Lean, we can declare a "type" of objects by writing `variable U : Type`. We can then declare a predicate on `U` by writing `variable P : U → Prop`. 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 :=
take x,
show P x, from sorry
```

Read `take 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 :=
take x,
show P x, from sorry

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

This constitutes the introduction rule for the universal quantifier.

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 $\forall x\ (even(x) \vee odd(x))$, we can conclude $even(t) \vee odd(t)$ for any term t. This motivates the elimination rule for the universal quantifier:

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

where t is an arbitrary term. In a Lean proof, this is implemented as follows:

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

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:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{HA : \forall x\ A(x)}{A(y)} \quad \dfrac{HB : \forall x\ B(x)}{B(y)}}{A(y) \wedge B(y)}}{\forall y\ (A(y) \wedge B(y))}}{\forall x\ B(x) \to \forall y\ (A(y) \wedge B(y))}\ HB}{\forall x\ A(x) \to (\forall x\ B(x) \to \forall y\ (A(y) \wedge B(y)))}\ HA$$

Notice that neither of the hypotheses HA or HB mention y, so that y is really "arbitrary" at the point where the universal quantifiers are introduced.

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,
take 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`
```

As an exercise, trying proving $(\forall x\ (A(x) \wedge B(x))) \to \forall x\ A(x)$ in natural deduction or Lean, or

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

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, or in Lean. For the latter, you need only replace each `sorry` below with a proof:

```
variable Person : Type
variable shaves : Person → Person → Prop
variable barber : Person
premise H : ∀ x, shaves barber x ↔ ¬ shaves x x

example : false :=
have H1 : shaves barber barber ↔ ¬ shaves barber barber, from sorry,
have H2 : ¬ shaves barber barber, from
  assume H2a : shaves barber barber,
  have H2b : ¬ shaves barber barber, from sorry,
  show false, from sorry,
have H3 : shaves barber barber, from sorry,
show false, from sorry
```

4.4 Some Number Theory

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:

$$\dfrac{\dfrac{}{even(n) \vee \neg even(n)} \qquad \dfrac{\dfrac{}{H_1 : even(n)}}{even(n) \vee odd(n)} \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{\forall n\ \neg even(n) \to odd(n)}}{\neg even(n) \to odd(n)} \qquad \dfrac{}{H_2 : \neg even(n)}}{odd(n)}}{even(n) \vee odd(n)}}{\dfrac{even(n) \vee odd(n)}{\forall n\ (even(n) \vee odd(n)}}\ {\scriptstyle H_1, H_2}$$

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

$$\dfrac{\dfrac{\dfrac{}{odd(n) \to \neg even(n)} \quad \dfrac{\dfrac{}{H : even(n) \wedge odd(n)}}{odd(n)}}{\neg even(n)} \quad \dfrac{\dfrac{}{H : even(n) \wedge odd(n)}}{even(n)}}{\dfrac{\dfrac{\bot}{\neg(even(n) \wedge odd(n))}\, H}{\forall n\, \neg(even(n) \wedge odd(n))}}$$

As we move from modeling basic rules of inference, however 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 basic 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.

In Lean's library, there are theorems `odd_of_not_even` and `even_of_not_odd`, whose uses are illustrated in the following:

```
import data.nat
open nat

example : ∀ n, ¬ even n → odd n :=
take n,
assume H : ¬ even n,
show odd n, from odd_of_not_even H

example : ∀ n, odd n → ¬ even n :=
take n,
assume H : odd n,
show ¬ even n, from not_even_of_odd H
```

Once again, notice the naming scheme: the conlusion is followed by the hypothesis, separated by the word `of`. Notice also that when applying the theorems, you do not need to specify the argument `n`: it is implicit in the hypothesis `H`. We can illustrate these theorems more concisely, by labeling `n` and `H` in the statement of the example:

```
example (n : ℕ) (H : ¬ even n) : odd n :=
odd_of_not_even H

example (n : ℕ) (H : odd n) : ¬ even n :=
not_even_of_odd H
```

In this text, we will often present theorems in the library in this way. Using these two theorems, the two facts we just proved in natural deduction can be proved in Lean as follows:

```
import data.nat
open nat classical
```

```
example : ∀ n, even n ∨ odd n :=
take n,
or.elim (em (even n))
  (suppose even n,
    show even n ∨ odd n, from or.inl this)
  (suppose ¬ even n,
    have odd n, from odd_of_not_even this,
    show even n ∨ odd n, from or.inr this)

example : ∀ n, ¬ (even n ∧ odd n) :=
take n,
assume H : even n ∧ odd n,
have even n, from and.left H,
have odd n, from and.right H,
have ¬ even n, from not_even_of_odd this,
show false, from `¬ even n` `even n`
```

Notice that we used the command `open classical` in order to use the law of the excluded middle, `em (even n)`, to split on cases.

Here are some more facts about parity that are found in the Lean library:

```
example (n : ℕ) (H : even n) : 2 ∣ n :=
dvd_of_even H

example (n : ℕ) (H : 2 ∣ n) : even n :=
even_of_dvd H

example (n : ℕ) : n ∣ n := dvd.refl n

example (k m n : ℕ) (H₁ : k ∣ m) (H₂ : m ∣ n) : k ∣ n :=
dvd.trans H₁ H₂

example (k m n : ℕ) (H₁ : k ∣ m) (H₂ : k ∣ n) : k ∣ m + n :=
dvd_add H₁ H₂

example (k m n : ℕ) (H₁ : k ∣ m + n) (H₂ : k ∣ m) : k ∣ n :=
dvd_of_dvd_add_left H₁ H₂

example (k m n : ℕ) (H₁ : k ∣ m + n) (H₂ : k ∣ n) : k ∣ m :=
dvd_of_dvd_add_right H₁ H₂

example : odd 1 :=
odd_one
```

To enter the "divides" symbol in Lean, you have to type `\|`. (The symbol is different from the plain `|` character.) Here are some examples of theorems that can be proved using these facts:

```
import data.nat
open nat

example : ∀ m n, even m → m ∣ n → even n :=
```

```
take m, take n,
suppose even m,
suppose m | n,
have 2 | m, from dvd_of_even `even m`,
have 2 | n, from dvd.trans this `m | n`,
show even n, from even_of_dvd this

example : ∀ m n, even m → even n → even (m + n) :=
take m, take n,
suppose even m,
suppose even n,
have 2 | m, from dvd_of_even `even m`,
have 2 | n, from dvd_of_even `even n`,
have 2 | m + n, from dvd_add `2 | m` `2 | n`,
show even (m + n), from even_of_dvd this

example : ∀ m n, even (m + n) → even m → even n :=
take m, take n,
suppose even (m + n),
suppose even m,
have 2 | m, from dvd_of_even `even m`,
have 2 | (m + n), from dvd_of_even `even (m + n)`,
have 2 | n, from dvd_of_dvd_add_left `2 | m + n` `2 | m`,
show even n, from even_of_dvd this

example : ∀ m n, even (m + n) → even n → even m :=
sorry

example : even 2 :=
have 2 | 2, from dvd.refl 2,
show even 2, from even_of_dvd this
```

The second-to-last one is left to you as an exercise. Remember, when you are trying to prove such theorems on your own, it is a good idea to prove them incrementally, using `sorry`. For example, for the first theorem, you might start as follows:

```
example : ∀ m n, even m → m | n → even n :=
take m, take n,
suppose even m,
suppose m | n,
show even n, from sorry
```

After checking to make sure that Lean accepts this, you can then add intermediate `have` statements, and so on.

If you wanted to use these theorems later on, you could name them:

```
import data.nat
open nat

theorem even_add_of_even_of_even : ∀ {m n}, even m → even n → even (m + n) :=
take m, take n,
suppose even m,
suppose even n,
```

```
have 2 ∣ m, from dvd_of_even `even m`,
have 2 ∣ n, from dvd_of_even `even n`,
have 2 ∣ m + n, from dvd_add `2 ∣ m` `2 ∣ n`,
show even (m + n), from even_of_dvd this

theorem even_of_even_add_left : ∀ {m n}, even (m + n) → even m → even n :=
take m, take n,
suppose even (m + n),
suppose even m,
have 2 ∣ m, from dvd_of_even `even m`,
have 2 ∣ (m + n), from dvd_of_even `even (m + n)`,
have 2 ∣ n, from dvd_of_dvd_add_left `2 ∣ m + n` `2 ∣ m`,
show even n, from even_of_dvd this
```

The curly braces around `m` and `n` in the first two theorems makes `m` and `n` *implicit arguments*, which means that you can write, for example, `even_add` H_1 H_2 for hypotheses H_1 `: even m` and H_2 `: even n`, rather than `even_add m n` H_1 H_2. In fact, the first of these is already in Lean's library:

```
import data.nat
open nat

check even_add_of_even_of_even
```

Using these, we can go on to prove the following:

```
example : ∀ n, even n → odd (n + 1) :=
take n,
suppose even n,
have ¬ even (n + 1), from
  suppose even (n + 1),
  have even 1, from even_of_even_add_left this `even n`,
  have ¬ even 1, from not_even_of_odd odd_one,
  show false, from `¬ even 1` `even 1`,
show odd (n + 1), from odd_of_not_even this

example : ∀ m n, even (m + n) → even n → even m :=
sorry

example : ∀ n, even (n + 1) → odd n :=
sorry
```

The last two are left for you to do as exercises.

Unfortunately, the facts we have presented to you so far do not let you prove that if `n` is odd, then `n+1` is even. Fortunately, that fact is also in the library (`succ` abbreviates "successor"), and you can use it to prove the second example below.

```
example (n : ℕ) (H : odd n) : even (n + 1) :=
even_succ_of_odd H
```

```
example (n : ℕ) (H : odd (n + 1)) : even n :=
sorry
```

Let us close with some examples of elementary theorems of number theory. (These are all exercises in Chapter 1 of *An Introduction to the Theory of Numbers* by Niven and Zuckerman.) For the moment, we will loosen up a bit and not insist that every fact we use can be proved axiomatically; let us take, as "common knowledge," facts such as these:

- A number is even if and only if it can be expressed in as $2n$, and odd if it can be expressed in the form $2n + 1$.
- A number is divisible by k if and only if it leaves a remainder of 0 when you divide it by k. In particular, of any k consecutive numbers $n, n + 1, n + 2, \ldots, n + (k - 1)$, at least one of them will be divisible by k.
- Expressed differently, if $k > 0$, then any natural number n can be expressed as $n = kq + r$, where $0 \leq r < k$.

The last fact is often known as the "quotient-remainder" theorem.

Theorem. The product of any three consecutive integers is divisible by 6.

Proof. Denote the three integers by n, $n + 1$, and $n + 2$. Then either n or $n + 1$ is divisible by 2, and either n, $n + 1$, or $n + 2$ is divisible by 3. So, their product is divisible by 6.

Theorem. For every n, $n^3 - n$ is divisible by 6.

Proof. We have $n^3 - n = (n - 1)n(n + 1)$, which is a product of three consecutive integers.

As exercises, try writing proving the following, informally:

- For any integer n, n^2 leaves a remainder of 0 or 1 when you divide it by 4. Hence $n^2 + 2$ is never divisible by 4.
- If n is odd, $n^2 - 1$ is divisible by 8.
- If m and n are odd, then $m^2 + n^2$ is even but not divisible by 4.
- Say that two integers "have the same parity" if they are both even or both odd. Prove that if m and n are any two integers, then $m + n$ and $m - n$ have the same parity.

4.5 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 $loves(x, y)$ to express that x loves y. In such a language, we might express the statement that "everyone loves someone" by writing $\forall x\ \exists y\ loves(x, y)$.

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 $loves(x, y)$ means that x is less than or equal to y. In that interpretation, the sentence

$$\forall x, y, z\ (loves(x, y) \wedge loves(y, z) \rightarrow loves(x, z))$$

is true, though in the original interpretation it makes an implausible claim about the nature of love triangles. In a later chapter, 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:

- $\forall x\ (woman(x) \rightarrow strong(x))$
- $\forall x\ (man(x) \rightarrow good{-}looking(x))$

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":

$$\forall x\ (prime(x) \wedge x \geq 2 \rightarrow odd(x)).$$

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." We will see how to do this in a later chapter.

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:

$$\forall p, q, L, M\ (point(p) \wedge point(q) \wedge line(L) \wedge line(M) \\ \wedge on(p, L) \wedge on(q, L) \wedge on(p, M) \wedge on(q, M) \rightarrow L = M)$$

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 $on(p, L)$ relating the two. Then the assertion above is rendered more simply as follows:

$$\forall p, q, L, M\ (on(p, L) \wedge on(q, L) \wedge on(p, M) \wedge on(q, M) \rightarrow L = M)$$

In Lean, we can model many-sorted logic by introducing a new type for each sort:

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

check ∀ (p q : Point) (L M : Line),
       on p L → on q L → on p M → 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,
       on p L → on q L → on p M → on q M → L = M
```

4.6 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.

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:

- $\{n \mid n \text{ is even}\}$
- $\{n \mid n \text{ is prime}\}$
- $\{n \mid n \text{ is prime and greater than } 2\}$
- $\{n \mid n \text{ can be written as a sum of squares}\}$
- $\{n \mid n \text{ is equal to 1, 2, or 3}\}$

This last set is written more simply $\{1, 2, 3\}$.

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. 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 to 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:

$$\begin{aligned}
&\forall x\ (x \notin \emptyset)\\
&\forall x\ (x \in \mathcal{U})\\
&\forall x\ (x \in A \cup B \leftrightarrow x \in A \vee x \in B)\\
&\forall x\ (x \in A \cap B \leftrightarrow x \in A \wedge x \in B)\\
&\forall x\ (x \in \overline{A} \leftrightarrow x \notin A)\\
&\forall x\ (x \in A \setminus B \leftrightarrow x \in A \wedge x \notin B)
\end{aligned}$$

The assertion that A is a subset of B can be written $\forall x\ (x \in A \rightarrow x \in B)$, 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. You should by now be able to discern this formal structure underlying *informal* proofs as well. Here are two examples.

Let $A, B, C, \ldots$ denotes sets of elements of some domain, X.

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

Proof. 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.

Theorem. $(A \setminus B) \setminus C = A \setminus (B \cup C)$.

Proof. 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 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 $(A \setminus B) \setminus C$.

You can carry out such reasoning in Lean, using methods you have already seen. For any type `X`, Lean gives us a type, `set X`, of sets of elements of `X`, with the element-of relation `x ∈ A`. We need only 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 {X : Type}
variables A B C : set X
```

We have made the type variable `X` implicit, because it can usually be inferred from context. The following pattern can be used to show that `A` is a subset of `B`:

```
example : A ⊆ B :=
take 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
  (take x,
    assume H : x ∈ A,
    show x ∈ B, from sorry)
  (take x,
    assume H : x ∈ B,
    show x ∈ A, from sorry)
```

Moreover, Lean supports the following nifty feature: all of the equivalences above are considered to hold "definitionally," which is to say, in most situations you can treat and the left- and right-hand-sides as being the same. In other words, you can act as though the expression `x ∈ A ∩ B` is no different from `x ∈ A ∧ x ∈ B`, and similarly for the other constructors.

```
example : ∀ x, x ∈ A → x ∈ B → x ∈ A ∩ B :=
take x,
suppose x ∈ A,
suppose x ∈ B,
show x ∈ A ∩ B, from and.intro `x ∈ A` `x ∈ B`

example : ∀ x : X, x ∉ ∅ :=
take x,
suppose x ∈ ∅,
show false, from this
```

In the second example, we annotated `x` with its type, `X`, because otherwise there is not enough information for Lean to infer which "empty set" we have in mind. You can type the symbols $\subseteq$, $\emptyset$, $\cup$, $\cap$, $\setminus$ as `\subeq` `\empty`, `\un`, `\i`, and `\\`, respectively. The universal set is denoted `univ`, and set complementation is denoted with a negation symbol.

The identifications above make it easy to prove some containment relations:

```
example : A \ B ⊆ A :=
take x,
suppose x ∈ A \ B,
show x ∈ A, from and.left this

example : A \ B ⊆ -B :=
take x,
suppose x ∈ A \ B,
have x ∉ B, from and.right this,
show x ∈ -B, from this
```

Here is the proof of the first identity that we proved informally above:

```
example : A ∩ (B ∪ C) = (A ∩ B) ∪ (A ∩ C) :=
eq_of_subset_of_subset
  (take 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`)
      (suppose x ∈ B,
        have x ∈ A ∩ B, from and.intro `x ∈ A` `x ∈ B`,
        show x ∈ (A ∩ B) ∪ (A ∩ C), from or.inl this)
      (suppose x ∈ C,
        have x ∈ A ∩ C, from and.intro `x ∈ A` `x ∈ C`,
        show x ∈ (A ∩ B) ∪ (A ∩ C), from or.inr this))
  (take x,
    suppose 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 above, because we have spelled out every last detail, though it may not be 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:

```
proposition inter_union_subset : A ∩ (B ∪ C) ⊆ (A ∩ B) ∪ (A ∩ C) :=
take 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`)
  (suppose x ∈ B,
    have x ∈ A ∩ B, from and.intro `x ∈ A` `x ∈ B`,
    show x ∈ (A ∩ B) ∪ (A ∩ C), from or.inl this)
  (suppose x ∈ C,
    have x ∈ A ∩ C, from and.intro `x ∈ A` `x ∈ C`,
    show x ∈ (A ∩ B) ∪ (A ∩ C), from or.inr this)

proposition inter_union_inter_subset : (A ∩ B) ∪ (A ∩ C) ⊆ A ∩ (B ∪ C) :=
take x,
suppose 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 propositions 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, `X`, but because the variable `X` was marked implicit, Lean figures it out from the context.

Bibliography