

How To Prove It with Lean

Daniel J. Velleman

Amherst College

© 2023–2025 Daniel J. Velleman.

Short excerpts from Daniel J. Velleman, *How To Prove It: A Structured Approach, 3rd Edition*
© Daniel J. Velleman 2019, published by Cambridge University Press, reprinted with permission.

Table of contents

Preface	1
About This Book	1
About Lean	2
Setting Up The HTPI Lean Package	2
VS Code and The HTPI Lean Package	4
License	5
Acknowledgments	5
1 Sentential Logic	6
2 Quantificational Logic	8
Introduction to Lean	9
A First Example	9
Term Mode	10
Tactic Mode	13
Types	17
3 Proofs	20
3.1 & 3.2. Proofs Involving Negations and Conditionals	20
3.3. Proofs Involving Quantifiers	27
3.4. Proofs Involving Conjunctions and Biconditionals	44
3.5. Proofs Involving Disjunctions	60
3.6. Existence and Uniqueness Proofs	73
3.7. More Examples of Proofs	90
4 Relations	108
4.1. Ordered Pairs and Cartesian Products	108
4.2. Relations	108
4.3. More About Relations	115
4.4. Ordering Relations	119
4.5. Equivalence Relations	125
5 Functions	133
5.1. Functions	133
5.2. One-to-One and Onto	139

Table of contents

5.3. Inverses of Functions	142
5.4. Closures	146
5.5. Images and Inverse Images: A Research Project	150
6 Mathematical Induction	154
6.1. Proof by Mathematical Induction	154
6.2. More Examples	163
6.3. Recursion	172
6.4. Strong Induction	183
6.5. Closures Again	192
7 Number Theory	199
7.1. Greatest Common Divisors	199
7.2. Prime Factorization	207
7.3. Modular Arithmetic	222
7.4. Euler’s Theorem	230
7.5. Public-Key Cryptography	245
8 Infinite Sets	251
8.1. Equinumerous Sets	251
8.1½. Debts Paid	275
8.2. Countable and Uncountable Sets	291
8.3. The Cantor–Schröder–Bernstein Theorem	303
Appendix	310
Tactics Used	310
Summary of Proof Techniques in Lean	311
Transitioning to Standard Lean	317
Typing Symbols	321

Preface

About This Book

This book is intended to accompany my book *How To Prove It: A Structured Approach, 3rd edition* (henceforth called *HTPI*), which is published by Cambridge University Press. Although this book is self-contained, we will sometimes have occasion to refer to passages in *HTPI*, so this book will be easiest to understand if you have a copy of *HTPI* available to you.

HTPI explains a systematic approach to constructing mathematical proofs. The purpose of this book is to show you how to use a computer software package called *Lean* to help you master the techniques presented in *HTPI*. *Lean* is free software that is available for Windows, MacOS, and Unix computers. It is also possible to run *Lean* in a web browser using GitHub Codespaces. We will explain below how to set up *Lean* on your computer; in a later chapter we'll explain how to get started using *Lean*.

The chapters and sections of this book are numbered to match the sections of *HTPI* to which they correspond. The first two chapters of *HTPI* cover preliminary topics in elementary logic and set theory that are needed to understand the proof techniques presented in later chapters. We assume that you are already familiar with that material (if not, go read those chapters in *HTPI*!), so Chapters 1 and 2 of this book will just briefly summarize the most important points. Those chapters are followed by an introduction to *Lean* that explains the basics of using *Lean* to write proofs. The presentation of proof techniques in *HTPI* begins in earnest in Chapter 3, so that is where we will begin to discuss how *Lean* can be used to master those techniques.

If you are reading this book online, then at the end of the title in the left margin you will find an icon that is a link to a pdf version of the book. Below that is a search box, which you can use to search for any word or phrase anywhere in the book. Below the search box is a list of the chapters of the book. Click on any chapter to go to that chapter. Within each chapter, a table of contents in the right margin lists the sections in that chapter. Again, you can go to any section by clicking on it. At the end of each chapter there are links to take you to the next or previous chapter.

About Lean

[Lean](#) is a kind of software package called a *proof assistant*. What that means is that Lean can help you to write proofs. As we will see over the course of this book, there are several ways in which Lean can be helpful. First of all, if you type a proof into Lean, then Lean can check the correctness of the proof and point out errors. As you are typing a proof into Lean, it will keep track of what has been accomplished so far in the proof and what remains to be done to finish the proof, and it will display that information for you. That can keep you moving in the right direction as you are figuring out a proof. And sometimes Lean can fill in small details of the proof for you.

Of course, to make this possible, you must type your proof in a format that Lean understands. Much of this book will be taken up with explaining how to write a proof so that Lean will understand it.

Note that this book uses a customized version of Lean. The customization is designed to make Lean proofs more readable and to bring Lean into closer agreement with *HTPI*. The appendix of this book includes [advice about transitioning from the Lean in this book to standard Lean](#).

Setting Up The HTPI Lean Package

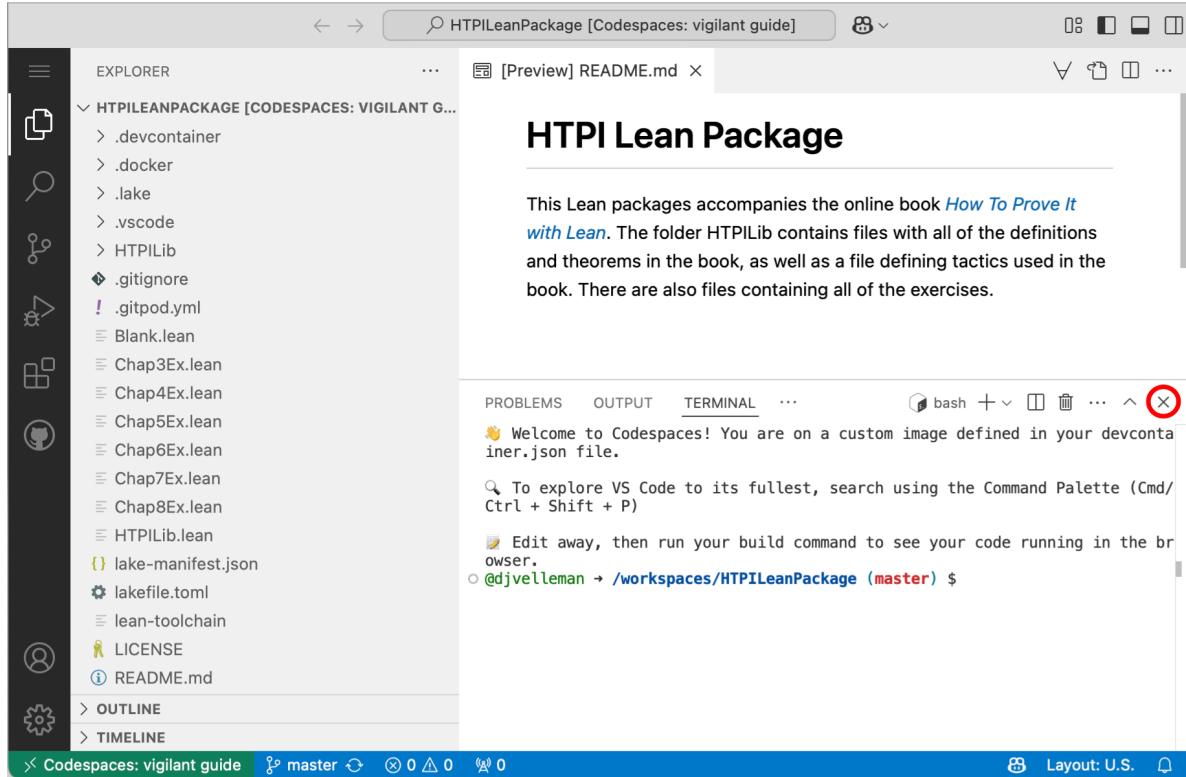
This book comes with a Lean “package” (or “project”), which is a folder containing a collection of files to be used with Lean. We will be using a software package called [Visual Studio Code](#) (VS Code) to access the files in the HTPI Lean Package. There are two ways to do this: by using GitHub Codespaces, or by installing Lean on your computer.

Using GitHub Codespaces

The easiest way to get started with Lean is to use GitHub Codespaces in your web browser. To open the HTPI Lean Package in Codespaces, click [here](#).

You will be prompted to create a GitHub account if you don’t already have one; a free account gives you 120 hours of use per month. Then you will be given various options for the creation of your codespace; you can use the default options. Click on the green “Create codespace” button. It will take several minutes to create your codespace, but this only needs to be done once; be patient. When that process is complete, you should see a window that looks something like this:

Setting Up The HTPI Lean Package



This image shows a version of VS Code, running in your codespace. You can access the menus for VS Code by clicking on the “≡” symbol at the top of the left sidebar. To the right of this sidebar, under the heading “Explorer,” you will see a list of the files and folders in the HTPI Lean Package. You can now click on the “X” on the right side of the lower part of the window (circled in red above) to close the terminal pane, since we won’t be using that pane.

Installing Lean

If you prefer to run Lean on your computer, you can find instructions for installing Lean [here](#). These instructions will lead you through installing VS Code, installing the Lean 4 extension in VS Code, and then opening the Lean 4 setup guide. The setup guide will then tell you how to install Lean dependencies and the Lean version manager. There is no need to consult the recommended books and documentation at this time—this book will tell you everything you need to know to use Lean with *HTPI*. (But later, if you want to learn more about Lean, you may find those resources useful.) And there is no need to follow the instructions for setting up a Lean project, because you will be using the HTPI Lean package.

You can download the HTPI Lean Package from <https://github.com/djvelleman/HTPILeanPackage>. After following the link, click on the green “Code” button and, in the pop-up menu,

select “Download ZIP”. Once the zip file has been downloaded, extract the contents of the file into a folder. You can put this folder wherever you want on your computer.

Next, in VS Code, select “Open Folder ...” from the File menu and open the folder containing the HTPI Lean Package that you downloaded. (If VS Code asks if you want to open the package in a “container”, it is probably best to say no.) Under the heading “Explorer” on the left side of the window, you should see a list of the files in the package. (If you don’t see the list, try clicking on the *Explorer* icon at the top of the left sidebar; it looks like this: .) Your screen should look similar to the image above in the section about using GitHub Codespaces. Click on the file “Blank.lean” in the file list. Then click on the “ \triangleright ” icon, and in the pop-up menu select “Project Actions ... > Project: Build Project”. Lean should “build” the HTPI Lean Package. You’ll see a number of messages, including some warnings that you can ignore. Eventually you should see the message “Project built successfully.” Your installation is now complete.

VS Code and The HTPI Lean Package

When you view the HTPI Lean Package in VS Code, you will see a list of the files and folders in the package under the “Explorer” heading. The folders are listed first, with a “ $>$ ” symbol in front of each; click on a folder to see the files inside it.

You won’t need to use most of the files in the package. In the chapter “Introduction to Lean,” you will learn how to edit the file “Blank.lean” to write your first proofs. The files “Chap3Ex.lean” through “Chap8Ex.lean” contain all of the exercises; you will enter your solutions to the exercises in those files. In the exercise file for a chapter, all Lean definitions and theorems from that chapter and all earlier chapters are available for use in solving the exercises.

The only other files that you may need to look at are in the HTPIlib folder. That folder includes files containing all Lean definitions and theorems in all the chapters, starting with “Introduction to Lean.” There is also a file in that folder that defines the customization mentioned earlier. All of these files are needed to make the package work. *Do not edit the files in the HTPIlib folder.*

Since you won’t be using the HTPI Lean Package right away, you can close it for now. If you opened it in GitHub Codespaces, then you can close the browser tab or window containing your codespace. GitHub will have given your codespace a whimsical name and saved it for you. When you are ready to start using Lean, you can return to your codespace from the GitHub Codespaces page at <https://github.com/codespaces>. If you installed VS Code and Lean on your computer, then you can quit VS Code and open it again when you are ready to start using Lean.

License

This book is licensed under a [Creative Commons Attribution-ShareAlike 4.0 International License](#). This license allows you to share or adapt the book. In any adaptation, you must identify Daniel J. Velleman as the author, and you must also acknowledge that excerpts from *How To Prove It, 3rd edition*, copyright Daniel J. Velleman 2019, published by Cambridge University Press, are reprinted with the permission of Cambridge University Press. Each such excerpt is identified in this book with a parenthetical note “(*HTPI* p. ...)” specifying the page of *How To Prove It, 3rd edition* from which the excerpt is taken. For further details, see the text of the [license](#).

Acknowledgments

A number of people have provided advice, encouragement, and feedback about this project. In particular, I would like to thank Jeremy Avigad, Clayton Cafiero, Nathan Carter, François Dorais, Charles Hoskinson, Heather Macbeth, Pietro Monticone, and Ketil Wright.

1 Sentential Logic

Chapter 1 of *How To Prove It* introduces the following symbols of logic:

Symbol	Meaning
\neg	not
\wedge	and
\vee	or
\rightarrow	if ... then
\leftrightarrow	iff (that is, if and only if)

As we will see, Lean uses the same symbols, with the same meanings. A statement of the form $P \wedge Q$ is called a *conjunction*, a statement of the form $P \vee Q$ is called a *disjunction*, a statement of the form $P \rightarrow Q$ is an *implication* or a *conditional* statement (with *antecedent* P and *consequent* Q), and a statement of the form $P \leftrightarrow Q$ is a *biconditional* statement. The statement $\neg P$ is the *negation* of P .

This chapter also establishes a number of logical equivalences that will be useful to us later:

Name		Equivalence	
De Morgan's Laws	$\neg(P \wedge Q)$	is equivalent to	$\neg P \vee \neg Q$
	$\neg(P \vee Q)$	is equivalent to	$\neg P \wedge \neg Q$
Double Negation Law	$\neg\neg P$	is equivalent to	P
Conditional Laws	$P \rightarrow Q$	is equivalent to	$\neg P \vee Q$
	$P \rightarrow Q$	is equivalent to	$\neg(P \wedge \neg Q)$
Contrapositive Law	$P \rightarrow Q$	is equivalent to	$\neg Q \rightarrow \neg P$

Finally, Chapter 1 of *HTPI* introduces some concepts from set theory. A *set* is a collection of objects; the objects in the collection are called *elements* of the set. The notation $x \in A$ means that x is an element of A . Two sets A and B are equal if they have exactly the same elements. We say that A is a *subset* of B , denoted $A \subseteq B$, if every element of A is an element of B . If $P(x)$ is a statement about x , then $\{x \mid P(x)\}$ denotes the set whose elements are the objects x for which $P(x)$ is true. And we have the following operations on sets:

1 Sentential Logic

$A \cap B = \{x \mid x \in A \wedge x \in B\}$ = the *intersection* of A and B ,

$A \cup B = \{x \mid x \in A \vee x \in B\}$ = the *union* of A and B ,

$A \setminus B = \{x \mid x \in A \wedge x \notin B\}$ = the *difference* of A and B ,

$A \Delta B = (A \setminus B) \cup (B \setminus A)$ = the *symmetric difference* of A and B .

2 Quantificational Logic

Chapter 2 of *How To Prove It* introduces two more symbols of logic, the quantifiers \forall and \exists . If $P(x)$ is a statement about an object x , then

$\forall x P(x)$ means “for all x , $P(x)$,”

and

$\exists x P(x)$ means “there exists some x such that $P(x)$.”

Lean also uses these symbols, although we will see that quantified statements are written slightly differently in Lean from the way they are written in *HTPI*. In the statement $P(x)$, the variable x is called a *free variable*. But in $\forall x P(x)$ or $\exists x P(x)$, it is a *bound variable*; we say that the quantifiers \forall and \exists *bind* the variable.

Once again, there are logical equivalences involving these symbols that will be useful to us later:

Quantifier Negation Laws		
$\neg \exists x P(x)$	is equivalent to	$\forall x \neg P(x)$
$\neg \forall x P(x)$	is equivalent to	$\exists x \neg P(x)$

Chapter 2 of *HTPI* also introduces some more advanced set theory operations. For any set A ,

$\mathcal{P}(A) = \{X \mid X \subseteq A\}$ = the *power set* of A .

Also, if \mathcal{F} is a family of sets—that is, a set whose elements are sets—then

$\bigcap \mathcal{F} = \{x \mid \forall A(A \in \mathcal{F} \rightarrow x \in A)\}$ = the *intersection* of the family \mathcal{F} ,

$\bigcup \mathcal{F} = \{x \mid \exists A(A \in \mathcal{F} \wedge x \in A)\}$ = the *union* of the family \mathcal{F} .

Finally, Chapter 2 introduces the notation $\exists!x P(x)$ to mean “there is exactly one x such that $P(x)$.” This can be thought of as an abbreviation for $\exists x(P(x) \wedge \neg \exists y(P(y) \wedge y \neq x))$. By the quantifier negation, De Morgan, and conditional laws, this is equivalent to $\exists x(P(x) \wedge \forall y(P(y) \rightarrow y = x))$.

Introduction to Lean

If you are reading this book in conjunction with *How To Prove It*, you should complete Section 3.2 of *HTPI* before reading this chapter. Once you have reached that point in *HTPI*, you are ready to start learning about Lean. In this chapter we'll explain the basics of writing proofs in Lean and getting feedback from Lean.

A First Example

We'll start with Example 3.2.4 in *How To Prove It*. Here is how the theorem and proof in that example appear in *HTPI* (*HTPI* p. 110; consult *HTPI* if you want to see how this proof was constructed):

Theorem. Suppose $P \rightarrow (Q \rightarrow R)$. Then $\neg R \rightarrow (P \rightarrow \neg Q)$.

Proof. Suppose $\neg R$. Suppose P . Since P and $P \rightarrow (Q \rightarrow R)$, it follows that $Q \rightarrow R$. But then, since $\neg R$, we can conclude $\neg Q$. Thus, $P \rightarrow \neg Q$. Therefore $\neg R \rightarrow (P \rightarrow \neg Q)$. \square

And here is how we would write the proof in Lean.¹

```
theorem Example_3_2_4
  (P Q R : Prop) (h : P → (Q → R)) : ¬R → (P → ¬Q) := by
  assume h2 : ¬R
  assume h3 : P
  have h4 : Q → R := h h3
  contrapos at h4           --Now h4 : ¬R → ¬Q
  show ¬Q from h4 h2
  done
```

¹Experienced Lean users will notice that the Lean proofs in this book look somewhat different from standard Lean proofs. This is because our proofs use a number of custom tactics. These tactics are designed to make our Lean proofs more readable and to bring Lean into closer agreement with *HTPI*. The custom tactics are defined in the file “*HTPIDefs.lean*”, which is in the folder “*HTPILib*” in the Lean package that accompanies this book. The appendix of this book includes [advice about transitioning from the Lean in this book to standard Lean](#).

Let's go through this Lean proof line-by-line and see what it means. The first line tells Lean that we are going to prove a theorem, and it gives the theorem a name, `Example_3_2_4`. The next line states the theorem. In the theorem as stated in *HTPI*, the letters P , Q , and R are used to stand for statements that are either true or false. In logic, such statements are often called *propositions*. The expression `(P Q R : Prop)` on the second line tells Lean that P , Q , and R will be used in this theorem to stand for propositions. The next parenthetical expression, `(h : P → (Q → R))`, states the hypothesis of the theorem and gives it the name h ; the technical term that Lean uses is that h is an *identifier* for the hypothesis. Assigning an identifier to the hypothesis gives us a way to refer to it when it is used later in the proof. Almost any string of characters that doesn't begin with a digit can be used as an identifier, but it is traditional to use identifiers beginning with the letter h for statements. After the statement of the hypothesis there is a colon followed by the conclusion of the theorem, `¬R → (P → ¬Q)`. Finally, at the end of the second line, the expression `:=` by signals the beginning of the proof.

Each of the remaining lines is a step in the proof. The first line of the proof introduces the assumption $\neg R$ and gives it the identifier $h2$. Of course, this corresponds precisely to the first sentence of the proof in *HTPI*. Similarly, the second line, corresponding to the second sentence of the *HTPI* proof, assigns the identifier $h3$ to the assumption P . The next line makes the inference $Q → R$, giving it the identifier $h4$. The inference is justified by combining statements h and $h3$ —that is, the statements $P → (Q → R)$ and P —exactly as in the third sentence of the proof in *HTPI*.

The next step of the proof in *HTPI* combines the statements $Q → R$ and $\neg R$ to draw the inference $\neg Q$. This reasoning is justified by the contrapositive law, which says that $Q → R$ is equivalent to its contrapositive, $\neg R → \neg Q$. In the Lean proof, this inference is broken up into two steps. In the fourth line of the proof, we ask Lean to rewrite statement $h4$ —that is, $Q → R$ —using the contrapositive law. Two hyphens in a row tell Lean that the rest of the line is a comment. Lean ignores comments and displays them in green. The comment on line four serves as a reminder that $h4$ now stands for the statement $\neg R → \neg Q$. Finally, in the last step of the proof, we combine the new $h4$ with $h2$ to infer $\neg Q$. There is no need to give this statement an identifier, because it completes the proof. In the proof in *HTPI*, there are a couple of final sentences explaining *why* this completes the proof, but Lean doesn't require this explanation.

Term Mode

Now that you have seen an example of a proof in Lean, it is time for you to write your first proof. Lean has two modes for writing proofs, called *term mode* and *tactic mode*. The example above was written in tactic mode, and that is the mode we will use for most proofs in this book. But before we study the construction of proofs in tactic mode, it will be helpful to learn a bit about term mode. Term mode is best for simple proofs, so we begin with a few very short proofs.

If you have not yet opened the HTPI Lean package in VS Code, either in GitHub Codespaces or on your computer, then go back and follow the instructions in the [preface](#). Once you have opened the package, click on the file Blank.lean in the list of files. The file starts with the line `import HTPILib.HTPIDefs`. Click on the blank line at the end of the file; this is where you will be typing your first proofs.

Now type in the following theorem and proof. (If you are reading this book online, then Lean examples like the one below will appear in gray boxes. You can copy the example to your clipboard by clicking in the upper-right corner of the box, and then you can paste it into a file in VS Code to try it out.)

```
theorem extremely_easy (P : Prop) (h : P) : P := h
```

If you have typed this correctly, Lean will put a check mark in the margin to the left of the theorem, indicating that the proof is correct. This theorem and proof are so short that we have put everything on one line. In this theorem, the letter `P` is used to stand for a proposition. The theorem has one hypothesis, `P`, which has been given the identifier `h`, and the conclusion of the theorem is also `P`. The notation `:=` indicates that what follows will be a proof in term mode.

Of course, the proof of the theorem is extremely easy: to prove `P`, we just have to point out that it is given as the hypothesis `h`. And so the proof in Lean consists of just one letter: `h`.

Even though this example is a triviality, there are some things to be learned from it. First of all, although we have been describing the letter `h` as an *identifier* for the hypothesis `P`, this example illustrates that Lean also considers `h` to be a *proof* of `P`. In general, when we see `h : P` in a Lean proof, where `P` is a proposition, we can think of it as meaning, not just that `h` is an identifier for the statement `P`, but also that `h` is a proof of `P`.

We can learn something else from this example by changing it slightly. If you change the final `h` to a different letter—say, `f`—you will see that Lean puts a red squiggly line under the `f`, like this:

```
theorem extremely_easy (P : Prop) (h : P) : P := f
```

This indicates that Lean has detected an error in the proof. Lean always indicates errors by putting a red squiggle under the offending text. Lean also puts a red X with a circle around it in the left margin next to the line with the error, and it puts a message in the Lean Infoview pane explaining what the error is. (If you don't see the Infoview pane on the right side of the window, click on the “ \forall ” icon near the top of the window and select “Infoview: Toggle Infoview” from the popup menu to make the Infoview pane appear.) In this case, the message is `unknown identifier 'f'`. The message is introduced by a heading, in red, that identifies the file, the line number, and the character position on that line where the error appears. If you

change f back to h , the red X with the circle around it, red squiggle, and error message go away, and Lean returns the check mark to the left margin next to the theorem.

Let's try a slightly less trivial example. You can type the next theorem below the previous one, leaving a blank line between them to keep them visually separate. To type the \rightarrow symbol in the next example, type `\to` and then hit either the space bar or the tab key; when you type either space or tab, the `\to` will change to \rightarrow . Alternatively, you can type `\r` (short for “right arrow”) or `\imp` (short for “implies”), again followed by either space or tab. Or, you can type `->`, and Lean will interpret it as \rightarrow . (There is a list in the appendix showing how to type all of the symbols used in this book.)

```
theorem very_easy
(P Q : Prop) (h1 : P → Q) (h2 : P) : Q := h1 h2
```

Indenting the second line is not necessary, but it is traditional. When stating a theorem, we will generally indent all lines after the first with two tabs in VS Code. Once you indent a line, VS Code will maintain that same indenting in subsequent lines until you delete tabs at the beginning of a line to reduce or eliminate indenting.

This time there are two hypotheses, $h1 : P \rightarrow Q$ and $h2 : P$. As explained in Section 3.2 of *HTPI*, the conclusion Q follows from these hypotheses by the logical rule *modus ponens*. To use modus ponens to complete this proof in term mode, we simply write the identifiers of the two hypotheses—which, as we have just seen, can also be thought of as proofs of the two hypotheses—one after the other, with a space between them. It is important to write the proof of the conditional hypothesis first, so the proof is written $h1\ h2$; if you try writing this proof as $h2\ h1$, you will get a red squiggle. In general, if a is a proof of any conditional statement $X \rightarrow Y$, and b is a proof of the antecedent X , then $a\ b$ is a proof of the consequent Y . The proofs a and b need not be simply identifiers; any proofs of a conditional statement and its antecedent can be combined in this way.

We'll try one more proof in term mode:

```
theorem easy (P Q R : Prop) (h1 : P → Q)
(h2 : Q → R) (h3 : P) : R :=
```

Note that in the statement of the theorem, you can break the lines however you please; this time we have put the declaration of P , Q , and R and the first hypothesis on the first line and the other two hypotheses on the second line. How can we prove the conclusion R ? Well, we have $h2 : Q \rightarrow R$, so if we could prove Q then we could use modus ponens to reach the desired conclusion. In other words, $h2\ _$ will be a proof of R , if we can fill in the blank with a proof of Q . Can we prove Q ? Yes, Q follows from $P \rightarrow Q$ and P by modus ponens, so $h1\ h3$ is a proof of Q . Filling in the blank, we conclude that $h2\ (h1\ h3)$ is a proof of R . Type it in, and you'll see that Lean will accept it. Note that the parentheses are important; if you write $h2\ h1\ h3$ then Lean will interpret it as $(h2\ h1)\ h3$, which doesn't make sense, and you'll get an error.

Tactic Mode

For more complicated proofs, it is easier to use tactic mode. Type the following theorem into Lean; to type the symbol \neg , type \not, followed again by either space or tab. Alternatively, if you type `Not P`, Lean will interpret it as meaning $\neg P$.

```
theorem two_imp (P Q R : Prop)
  (h1 : P → Q) (h2 : Q → ¬R) : R → ¬P :=
```

Lean is now waiting for you to type a proof in term mode. To switch to tactic mode, type by after `:=`. We find it helpful to set off a tactic proof from the surrounding text by indenting it with one tab, and also by marking where the proof ends. To do this, leave a blank line after the statement of the theorem, adjust the indenting to one tab, and type `done`. You will type your proof between the statement of the theorem and the line containing `done`, so click on the blank line between them to position the cursor there. Lean can be fussy about indenting; it will be important to indent all steps of the proof by the same amount.

One of the advantages of tactic mode is that Lean displays, in the Lean Infoview pane, information about the status of the proof as you write it. As soon as you position your cursor on the blank line, Lean displays what it calls the “tactic state” in the Infoview pane. Your screen should look like this. (If you are reading this book online, then examples like the one below may not display well if your window is too narrow; try adjusting the width of the window.)

Lean File

```
theorem two_imp (P Q R : Prop)
  (h1 : P → Q) (h2 : Q → ¬R) : R → ¬P := by
done
```

Tactic State in Infoview

P Q R : Prop
h1 : P → Q
h2 : Q → ¬R
↑ R → ¬P

The red squiggle under `done` indicates that Lean knows that the proof isn’t done. The tactic state in the Infoview pane is very similar to the lists of givens and goals that are used in *HTPI*. The hypotheses `h1 : P → Q` and `h2 : Q → ¬R` are examples of what are called *givens* in *HTPI*. The tactic state above says that `P`, `Q`, and `R` stand for propositions, and then it lists the two givens `h1` and `h2`. The symbol `↑` in the last line labels the *goal*, `R → ¬P`. The tactic state is a valuable tool for guiding you as you are figuring out a proof; whenever you are trying to decide on the next step of a proof, you should look at the tactic state to see what givens you have to work with and what goal you need to prove.

From the givens `h1` and `h2` it shouldn’t be hard to prove `P → ¬R`, but the goal is `R → ¬P`. This suggests that we should prove the contrapositive of the goal. Type `contrapos` (indented by one tab, to match the indenting of `done`) to tell Lean that you want to replace the goal with its contrapositive. As soon as you type `contrapos`, Lean will update the tactic state to reflect the change in the goal. You should now see this:

Lean File

```
theorem two_imp (P Q R : Prop)
  (h1 : P → Q) (h2 : Q → ¬R) : R → ¬P := by
  contrapos
  done
```

Tactic State in Infoview

P	Q	R	: Prop
h1	:	P	→ Q
h2	:	Q	→ ¬R
↑ P → ¬R			

If you want to make your proof a little more readable, you could add a comment saying that the goal has been changed to $P \rightarrow \neg R$. To prove the new goal, we will assume P and prove $\neg R$. So type `assume h3 : P` on a new line (after `contrapos`, but before `done`). Once again, the tactic state is immediately updated. Lean adds $h3 : P$ as a new given, and it knows, without having to be told, that the goal should now be $\neg R$:

Lean File

```
theorem two_imp (P Q R : Prop)
  (h1 : P → Q) (h2 : Q → ¬R) : R → ¬P := by
  contrapos          --Goal is now P → ¬R
  assume h3 : P
  done
```

Tactic State in Infoview

P	Q	R	: Prop
h1	:	P	→ Q
h2	:	Q	→ ¬R
h3	:	P	
↑ ¬R			

We can now use modus ponens to infer Q from $h1 : P \rightarrow Q$ and $h3 : P$. As we saw earlier, this means that $h1 \ h3$ is a term-mode proof of Q . So on the next line, type `have h4 : Q := h1 h3`. To make an inference, you need to provide a justification, so `:=` here is followed by the term-mode proof of Q . Usually we will use `have` to make easy inferences for which we can give simple term-mode proofs. (We'll see later that it is also possible to use `have` to make an inference justified by a tactic-mode proof.) Of course, Lean updates the tactic state by adding the new given $h4 : Q$:

Lean File

```
theorem two_imp (P Q R : Prop)
  (h1 : P → Q) (h2 : Q → ¬R) : R → ¬P := by
  contrapos          --Goal is now P → ¬R
  assume h3 : P
  have h4 : Q := h1 h3
  done
```

Tactic State in Infoview

P	Q	R	: Prop
h1	:	P	→ Q
h2	:	Q	→ ¬R
h3	:	P	
h4	:	Q	
↑ ¬R			

Finally, to complete the proof, we can infer the goal $\neg R$ from $h2 : Q \rightarrow \neg R$ and $h4 : Q$, using the term-mode proof $h2 \ h4$. Type `show ¬R from h2 h4` to complete the proof. You'll notice several changes in the display: the error will disappear from the word `done`, a check mark will appear in the left margin next to the theorem, and the tactic state will say "No goals" to indicate that there is nothing left to prove:

Lean File

```
theorem two_imp (P Q R : Prop)
  (h1 : P → Q) (h2 : Q → ¬R) : R → ¬P := by
  contrapos          --Goal is now P → ¬R
  assume h3 : P
  have h4 : Q := h1 h3
  show ¬R from h2 h4
  done
```

Tactic State in Infoview

No goals

Congratulations! You've written your first proof in tactic mode. If you move your cursor around in the proof, you will see that Lean always displays in the Infoview the tactic state at the point in the proof where the cursor is located. Try clicking on different lines of the proof to see how the tactic state changes over the course of the proof. If you want to try another example, you could try typing in the first example in this chapter. You will learn the most from this book if you continue to type the examples into Lean and see for yourself how the tactic state gets updated as the proof is written.

In each step of a tactic-mode proof, we invoke a *tactic*. In the proofs above, we have used four tactics: `contrapos`, `assume`, `have`, and `show`. If the goal is a conditional statement, the `contrapos` tactic replaces it with its contrapositive. If `h` is a given that is a conditional statement, then `contrapos at h` will replace `h` with its contrapositive. If the goal is a conditional statement $P \rightarrow Q$, you can use the `assume` tactic to assume the antecedent P , and Lean will set the goal to be the consequent Q . You can use the `have` tactic to make an inference from your givens, as long as you can justify the inference with a proof. The `show` tactic is similar, but it is used to infer the goal, thus completing the proof. And we have learned how to use one rule of inference in term mode: modus ponens. In the rest of this book we will learn about other tactics and other term-mode rules.

Before continuing, it might be useful to summarize how you type statements into Lean. We have already told you how to type the symbols \rightarrow and \neg , but you will want to know how to type all of the logical connectives. In each case, the command to produce the symbol must be followed by space or tab, but there is also a plain text alternative:

Symbol	How To Type It	Plain Text Alternative
\neg	<code>\not</code> or <code>\n</code>	Not
\wedge	<code>\and</code>	<code>/\</code>
\vee	<code>\or</code> or <code>\v</code>	<code>\vee</code>
\rightarrow	<code>\to</code> or <code>\r</code> or <code>\imp</code>	<code>-></code>
\leftrightarrow	<code>\iff</code> or <code>\lr</code>	<code><-></code>

Lean has conventions that it follows to interpret a logical statement when there are not enough parentheses to indicate how terms are grouped in the statement. For our purposes, the most

Tactic Mode

important of these conventions is that $P \rightarrow Q \rightarrow R$ is interpreted as $P \rightarrow (Q \rightarrow R)$, not $(P \rightarrow Q) \rightarrow R$. The reason for this is simply that statements of the form $P \rightarrow (Q \rightarrow R)$ come up much more often in proofs than statements of the form $(P \rightarrow Q) \rightarrow R$. (Lean also follows this “grouping-to-the-right” convention for \wedge and \vee , although this makes less of a difference, since these connectives are associative.) Of course, when in doubt about how to type a statement, you can always put in extra parentheses to avoid confusion.

We will be using tactics to apply several logical equivalences. Here are tactics corresponding to all of the [logical laws](#) listed in Chapter 1, as well as one additional law:

Logical Law	Tactic		Transformation	
Contrapositive Law	contrapos	$P \rightarrow Q$	is changed to	$\neg Q \rightarrow \neg P$
De Morgan's Laws	demorgan	$\neg(P \wedge Q)$	is changed to	$\neg P \vee \neg Q$
		$\neg(P \vee Q)$	is changed to	$\neg P \wedge \neg Q$
		$P \wedge Q$	is changed to	$\neg(\neg P \vee \neg Q)$
		$P \vee Q$	is changed to	$\neg(\neg P \wedge \neg Q)$
Conditional Laws	conditional	$P \rightarrow Q$	is changed to	$\neg P \vee Q$
		$\neg(P \rightarrow Q)$	is changed to	$P \wedge \neg Q$
		$P \vee Q$	is changed to	$\neg P \rightarrow Q$
		$P \wedge Q$	is changed to	$\neg(P \rightarrow \neg Q)$
Double Negation Law	double_neg	$\neg\neg P$	is changed to	P
Biconditional Negation Law	bicond_neg	$\neg(P \leftrightarrow Q)$	is changed to	$\neg P \leftrightarrow Q$
		$P \leftrightarrow Q$	is changed to	$\neg(\neg P \leftrightarrow Q)$

All of these tactics work the same way as the `contrapos` tactic: by default, the transformation is applied to the goal; to apply it to a given `h`, add at `h` after the tactic name.

Types

All of our examples so far have just used letters to stand for propositions. To prove theorems with mathematical content, we will need to introduce one more idea.

The underlying theory on which Lean is based is called *type theory*. We won't go very deeply into type theory, but we will need to make use of the central idea of the theory: every variable in Lean must have a type. What this means is that, when you introduce a variable to stand for a mathematical object in a theorem or proof, you must specify what type of object the variable stands for. We have already seen this idea in action: in our first example, the expression `(P Q R : Prop)` told Lean that the variables `P`, `Q`, and `R` have type `Prop`, which means they stand for propositions. There are types for many kinds of mathematical objects. For example, `Nat` is the type of natural numbers, and `Real` is the type of real numbers. So if you want to state a theorem about real numbers `x` and `y`, the statement of your theorem might start with `(x y : Real)`. You must include such a type declaration before you can use the variables `x` and `y` as free variables in the hypotheses or conclusion of your theorem.

What about sets? If you want to prove a theorem about a set `A`, can you say that `A` has type `Set`? No, Lean is fussier than that. Lean wants to know, not only that `A` is a set, but also what the type of the elements of `A` is. So you can say that `A` has type `Set Nat` if `A` is a set whose elements are natural numbers, or `Set Real` if it is a set of real numbers, or even `Set (Set Nat)` if it is a set whose elements are sets of natural numbers. Here is an example of a simple

Types

theorem about sets; it is a simplified version of Example 3.2.5 in *HTPI*. To type the symbols \in , \notin , and \setminus in this theorem, type `\in`, `\notin`, and `\setminus`, respectively.

Lean File

```
theorem Example_3_2_5_simple
  (B C : Set Nat) (a : Nat)
  (h1 : a ∈ B) (h2 : a ∉ B \ C) : a ∈ C := by
  done
```

Tactic State in Infoview

```
B C : Set ℕ
a : ℕ
h1 : a ∈ B
h2 : a ∉ B \ C
↑ a ∈ C
```

The second line of this theorem statement declares that the variables B and C stand for sets of natural numbers, and a stands for a natural number. The third line states the two hypotheses of the theorem, $a \in B$ and $a \notin B \setminus C$, and the conclusion, $a \in C$. (Note that Lean occasionally writes things slightly differently in the tactic state. In this case, Lean has written \mathbb{N} instead of `Nat`.)

To figure out this proof, we'll imitate the reasoning in Example 3.2.5 in *HTPI*. We begin by writing out the meaning of the given $h2$. Fortunately, we have a tactic for that. The tactic `define` writes out the definition of the goal, and as usual we can add `at` to apply the tactic to a given rather than the goal. Here's the situation after using the tactic `define at h2`:

Lean File

```
theorem Example_3_2_5_simple
  (B C : Set Nat) (a : Nat)
  (h1 : a ∈ B) (h2 : a ∉ B \ C) : a ∈ C := by
  define at h2      --Now h2 : ¬(a ∈ B ∧ a ∉ C)
  done
```

Tactic State in Infoview

```
B C : Set ℕ
a : ℕ
h1 : a ∈ B
h2 : ¬(a ∈ B ∧ a ∉ C)
↑ a ∈ C
```

Looking at the tactic state, we see that Lean has written out the meaning of set difference in $h2$. And now we can see that, as in Example 3.2.5 in *HTPI*, we can put $h2$ into a more useful form by applying first one of De Morgan's laws to rewrite it as $a \notin B \vee a \in C$ and then a conditional law to change it to $a \in B \rightarrow a \in C$:

Lean File

```
theorem Example_3_2_5_simple
  (B C : Set Nat) (a : Nat)
  (h1 : a ∈ B) (h2 : a ∉ B \ C) : a ∈ C := by
  define at h2      --Now h2 : ¬(a ∈ B ∧ a ∉ C)
  demorgan at h2   --Now h2 : a ∉ B ∨ a ∈ C
  conditional at h2 --Now h2 : a ∈ B → a ∈ C
  done
```

Tactic State in Infoview

```
B C : Set ℕ
a : ℕ
h1 : a ∈ B
h2 : a ∈ B → a ∈ C
↑ a ∈ C
```

Occasionally, you may feel that the application of two tactics one after the other should be thought of as a single step. To allow for this, Lean lets you put two tactics on the same line, separated by a semicolon. For example, in this proof you could write the use of De Morgan's law and the conditional law as a single step by writing `demorgan at h2; conditional at h2`. Now the rest is easy: we can apply modus ponens to reach the goal:

Lean File

```
theorem Example_3_2_5_simple
  (B C : Set Nat) (a : Nat)
  (h1 : a ∈ B) (h2 : a ∉ B \ C) : a ∈ C := by
  define at h2      --Now h2 : ¬(a ∈ B ∧ a ∉ C)
  demorgan at h2; conditional at h2
  --Now h2 : a ∈ B → a ∈ C
  show a ∈ C from h2 h1
  done
```

Tactic State in Infoview

No goals

There is one unfortunate feature of this theorem: We have stated it as a theorem about sets of natural numbers, but the proof has nothing to do with natural numbers. Exactly the same reasoning would prove a similar theorem about sets of real numbers, or sets of objects of any other type. Do we need to write a different theorem for each of these cases? No, fortunately there is a way to write one theorem that covers all the cases:

```
theorem Example_3_2_5_simple_general
  (U : Type) (B C : Set U) (a : U)
  (h1 : a ∈ B) (h2 : a ∉ B \ C) : a ∈ C := by
```

In this version of the theorem, we have introduced a new variable `U`, whose type is ... `Type!` So `U` can stand for any type. You can think of the variable `U` as playing the role of the universe of discourse, an idea that was introduced in Section 1.3 of *HTPI*. The sets `B` and `C` contain elements from that universe of discourse, and `a` belongs to the universe. You can prove the new version of the theorem by using exactly the same sequence of tactics as before.

3 Proofs

3.1 & 3.2. Proofs Involving Negations and Conditionals

Sections 3.1 and 3.2 of *How To Prove It* present strategies for dealing with givens and goals involving negations and conditionals. We restate those strategies here, and explain how to use them with Lean.

Section 3.1 gives two strategies for proving a goal of the form $P \rightarrow Q$ (*HTPI* pp. 95, 96):

To prove a goal of the form $P \rightarrow Q$:

1. Assume P is true and prove Q .
2. Assume Q is false and prove that P is false.

We've already seen how to carry out both of these strategies in Lean. For the first strategy, use the `assume` tactic to introduce the assumption P and assign an identifier to it; Lean will automatically set Q as the goal. We can summarize the effect of using this strategy by showing how the tactic state changes if you use the tactic `assume h : P`:

Tactic State Before Using Strategy

```
:  
⊢ P → Q
```

Tactic State After Using Strategy

```
:  
h : P  
⊢ Q
```

The second strategy is justified by the contrapositive law. In Lean, you can use the `contrapos` tactic to rewrite the goal as $\neg Q \rightarrow \neg P$ and then use the tactic `assume h : \neg Q`. The net effect of these two tactics is:

Tactic State Before Using Strategy

```
:  
⊢ P → Q
```

Tactic State After Using Strategy

```
:  
h : ¬Q  
⊢ ¬P
```

Section 3.2 gives two strategies for using givens of the form $P \rightarrow Q$, with the second once again being a variation on the first based on the contrapositive law (*HTPI* p. 108):

To use a given of the form $P \rightarrow Q$:

1. If you are also given P , or you can prove that P is true, then you can use this given to conclude that Q is true.
2. If you are also given $\neg Q$, or you can prove that Q is false, then you can use this given to conclude that P is false.

The first strategy is the modus ponens rule of inference, and we saw in the last chapter that if you have $h1 : P \rightarrow Q$ and $h2 : P$, then $h1 h2$ is a (term-mode) proof of Q ; often we use this rule with the `have` or `show` tactic. For the second strategy, if you have $h1 : P \rightarrow Q$ and $h2 : \neg Q$, then the `contrapos at h1` tactic will change $h1$ to $h1 : \neg Q \rightarrow \neg P$, and then $h1 h2$ will be a proof of $\neg P$.

All of the strategies listed above for working with conditional statements as givens or goals were illustrated in examples in the last chapter.

Section 3.2 of *HTPI* offers two strategies for proving negative goals (*HTPI* pp. 101, 102):

To prove a goal of the form $\neg P$:

1. Reexpress the goal in some other form.
2. Use proof by contradiction: assume P is true and try to deduce a contradiction.

For the first strategy, the tactics `demorgan`, `conditional`, `double_neg`, and `bicond_neg` may be useful, and we saw how those tactics work in the last chapter. But how do you write a proof by contradiction in Lean? The answer is to use a tactic called `by_contra`. If the goal is $\neg P$, then the tactic `by_contra h` will introduce the assumption $h : P$ and set the goal to be `False`, like this:

Tactic State Before Using Strategy

```
:  
↑ ⊥
```

Tactic State After Using Strategy

```
:  
h : P  
↑ False
```

In Lean, `False` represents a statement that is always false—that is, a contradiction, as that term is defined in Section 1.2 of *HTPI*. The `by_contra` tactic can actually be used even if the goal is not a negative statement. If the goal is a statement P that is not a negative statement, then `by_contra h` will initiate a proof by contradiction by introducing the assumption $h : \neg P$ and setting the goal to be `False`.

You will usually complete a proof by contradiction by deducing two contradictory statements—say, $h1 : Q$ and $h2 : \neg Q$. But how do you convince Lean that the proof is over? You must be able to prove the goal `False` from the two givens $h1$ and $h2$. There are two ways to do this. The first is based on the fact that Lean treats a statement of the form $\neg Q$ as meaning the same

3.1 & 3.2. Proofs Involving Negations and Conditionals

thing as $Q \rightarrow \text{False}$. This makes sense, because these statements are logically equivalent, as shown by the following truth table:

Q	$\neg Q$	$(Q \rightarrow \text{False})$
F	T	F
T	F	F

Thinking of $h2 : \neg Q$ as meaning $h2 : Q \rightarrow \text{False}$, we can combine it with $h1 : Q$ using modus ponens to deduce False . In other words, $h2 \ h1$ is a proof of False .

But there is a second way of completing the proof that it is worthwhile to know about. From contradictory statements $h1 : Q$ and $h2 : \neg Q$ you can validly deduce *any* statement. This follows from the definition of a *valid argument* in Section 1.1 of *HTPI*. According to that definition, you can validly infer a conclusion R from premises $h1 : Q$ and $h2 : \neg Q$ if the premises cannot both be true without the conclusion also being true. In this case, that standard is met, for the simple reason that the premises cannot both be true! (This gives part of the answer to exercise 18 in Section 1.2 of *HTPI*.) Thus, Lean has a rule that allows you to prove any statement from contradictory premises. If you have $h1 : Q$ and $h2 : \neg Q$, then Lean will recognize `absurd` $h1 \ h2$ as a (term-mode) proof of *any* statement.

To summarize, if you have $h1 : Q$ and $h2 : \neg Q$, then there are two ways to prove False . Lean will recognize $h2 \ h1$ as a proof of False , and it will recognize `absurd` $h1 \ h2$ as a proof of any statement, including False . Notice the difference in the order in which $h1$ and $h2$ are listed in these two proofs: In the first one, the negative statement $h2$ must come first, just as the conditional statement must come first in an application of modus ponens. But in a proof using `absurd`, the negative statement must come second.

To illustrate proof by contradiction in Lean, let's redo our first example from the last Chapter in a different way. That example was based on Example 3.2.4 in *HTPI*. We'll begin with the same first two steps, introducing two assumptions.

Lean File

```
theorem Example_3_2_4_v2 (P Q R : Prop)
  (h : P → (Q → R)) : ¬R → (P → ¬Q) := by
  assume h2 : ¬R
  assume h3 : P
  done
```

Tactic State in Infoview

P	Q	R	: Prop
h	:	P	\rightarrow Q \rightarrow R
h2	:	¬R	
h3	:	P	
$\vdash \neg Q$			

Now the goal is a negative statement, so we use the tactic `by_contra h4` to introduce the assumption $h4 : Q$ and set the goal to be False :

Lean File

```
theorem Example_3_2_4_v2 (P Q R : Prop)
  (h : P → (Q → R)) : ¬R → (P → ¬Q) := by
  assume h2 : ¬R
  assume h3 : P
  by_contra h4
  done
```

Tactic State in Infoview

```
P Q R : Prop
h : P → Q → R
h2 : ¬R
h3 : P
h4 : Q
↑ False
```

Using the givens h , $h3$, and $h4$ we can deduce first $Q \rightarrow R$ and then R by two applications of modus ponens:

Lean File

```
theorem Example_3_2_4_v2 (P Q R : Prop)
  (h : P → (Q → R)) : ¬R → (P → ¬Q) := by
  assume h2 : ¬R
  assume h3 : P
  by_contra h4
  have h5 : Q → R := h h3
  have h6 : R := h5 h4
  done
```

Tactic State in Infoview

```
P Q R : Prop
h : P → Q → R
h2 : ¬R
h3 : P
h4 : Q
h5 : Q → R
h6 : R
↑ False
```

Now we have a contradiction: $h2 : \neg R$ and $h6 : R$. To complete the proof, we deduce `False` from these two givens. Either $h2$ $h6$ or absurd $h6$ $h2$ would be accepted by Lean as a proof of `False`:

Lean File

```
theorem Example_3_2_4_v2 (P Q R : Prop)
  (h : P → (Q → R)) : ¬R → (P → ¬Q) := by
  assume h2 : ¬R
  assume h3 : P
  by_contra h4
  have h5 : Q → R := h h3
  have h6 : R := h5 h4
  show False from h2 h6
  done
```

Tactic State in Infoview

```
No goals
```

Finally, we have two strategies for using a given that is a negative statement (*HTPI* pp. 105, 108):

To use a given of the form $\neg P$:

1. Reexpress the given in some other form.
2. If you are doing a proof by contradiction, you can achieve a contradiction by proving P , since that would contradict the given $\neg P$.

Of course, strategy 1 suggests the use of the `demorgan`, `conditional`, `double_neg`, and `bicond_neg` tactics, if they apply. For strategy 2, if you are doing a proof by contradiction and you have a given $h : \neg P$, then the tactic `contradict h` will set the goal to be P , which will complete the proof by contradicting h . In fact, this tactic can be used with any given; if you have a given $h : P$, where P is not a negative statement, then `contradict h` will set the goal to be $\neg P$. You can also follow the word `contradict` with a proof that is more complicated than a single identifier. For example, if you have givens $h1 : P \rightarrow \neg Q$ and $h2 : P$, then $h1 h2$ is a proof of $\neg Q$, so the tactic `contradict h1 h2` will set the goal to be Q .

If you're not doing a proof by contradiction, then the tactic `contradict h with h'` will first initiate a proof by contradiction by assuming the negation of the goal, giving that assumption the identifier h' , and then it will set the goal to be the negation of the statement proven by h . In other words, `contradict h with h'` is shorthand for `by_contra h'; contradict h`.

We can illustrate this with yet another way to write the proof from Example 3.2.4. Our first three steps will be the same as last time:

Lean File

```
theorem Example_3_2_4_v3 (P Q R : Prop)
  (h : P → (Q → R)) : ¬R → (P → ¬Q) := by
  assume h2 : ¬R
  assume h3 : P
  by_contra h4
  done
```

Tactic State in Infoview

P	Q	R	: Prop
h	:	$P \rightarrow Q \rightarrow R$	
$h2$:	$\neg R$	
$h3$:	P	
$h4$:	Q	
↑ False			

Since we are now doing a proof by contradiction and the given $h2 : \neg R$ is a negative statement, a likely way to proceed is to try to prove R , which would contradict $h2$. So we use the tactic `contradict h2`:

Lean File

```
theorem Example_3_2_4_v3 (P Q R : Prop)
  (h : P → (Q → R)) : ¬R → (P → ¬Q) := by
  assume h2 : ¬R
  assume h3 : P
  by_contra h4
  contradict h2
  done
```

Tactic State in Infoview

P	Q	R	: Prop
h	:	$P \rightarrow Q \rightarrow R$	
$h2$:	$\neg R$	
$h3$:	P	
$h4$:	Q	
↑ R			

3.1 & 3.2. Proofs Involving Negations and Conditionals

As before, we can now prove R by combining h , $h3$, and $h4$. In fact, we could do it in one step: by modus ponens, $h \ h3$ is a proof of $Q \rightarrow R$, and therefore, by another application of modus ponens, $(h \ h3) \ h4$ is a proof of R . The parentheses here are not necessary; Lean will interpret $h \ h3 \ h4$ as $(h \ h3) \ h4$, so we can complete the proof like this:

Lean File

```
theorem Example_3_2_4_v3 (P Q R : Prop)
  (h : P → (Q → R)) : ¬R → (P → ¬Q) := by
  assume h2 : ¬R
  assume h3 : P
  by_contra h4
  contradict h2
  show R from h h3 h4
  done
```

Tactic State in Infoview

No goals

You could shorten this proof slightly by replacing the lines `by_contra h4` and `contradict h2` with the single line `contradict h2 with h4`.

There is one more idea that is introduced in Section 3.2 of *HTPI*. The last example in that section illustrates how you can sometimes use rules of inference to work backwards. Here's a similar example in Lean:

Lean File

```
theorem Like_Example_3_2_5
  (U : Type) (A B C : Set U) (a : U)
  (h1 : a ∈ A) (h2 : a ∉ A \ B)
  (h3 : a ∈ B → a ∈ C) : a ∈ C := by
  done
```

Tactic State in Infoview

U : Type
A B C : Set U
a : U
h1 : a ∈ A
h2 : a ∉ A \ B
h3 : a ∈ B → a ∈ C
 $\vdash a \in C$

The goal is $a \in C$, and the only given that even mentions C is $h3 : a \in B \rightarrow a \in C$. If only we could prove $a \in B$, then we could apply $h3$, using modus ponens, to reach our goal. So it would make sense to work toward the goal of proving $a \in B$.

To get Lean to use this proof strategy, we use the tactic `apply h3 _`. The underscore here represents a blank to be filled in by Lean. You might think of this tactic as asking Lean the question: If we want $h3 _$ to be a proof of the goal $a \in C$, what do we have to put in the blank? Lean is able to figure out that the answer is: a proof of $a \in B$. So it sets the goal to be $a \in B$, since a proof of that goal, when inserted into the blank in $h3 _$, would prove the original goal $a \in C$:

Lean File

```
theorem Like_Example_3_2_5
  (U : Type) (A B C : Set U) (a : U)
  (h1 : a ∈ A) (h2 : a ∉ A \ B)
  (h3 : a ∈ B → a ∈ C) : a ∈ C := by
  apply h3 _
  done
```

Tactic State in Infoview

```
U : Type
A B C : Set U
a : U
h1 : a ∈ A
h2 : a ∉ A \ B
h3 : a ∈ B → a ∈ C
↑ a ∈ B
```

Our situation now is very much like the one in the theorem `Example_3_2_5_Simple` in the previous chapter, and the rest of our proof will be similar to the proof there. The given `h2` is a negative statement (`a ∉ A \ B` is shorthand for $\neg a \in A \setminus B$), so, as suggested by our first strategy for using negative givens, we reexpress it as an equivalent positive statement. Writing out the definition of set difference, `h2` means $\neg(a \in A \wedge a \notin B)$, and then one of De Morgan's laws and a conditional law allow us to rewrite it first as $a \notin A \vee a \in B$ and then as $a \in A \rightarrow a \in B$. Of course, we have tactics to accomplish all of these reexpressions:

Lean File

```
theorem Like_Example_3_2_5
  (U : Type) (A B C : Set U) (a : U)
  (h1 : a ∈ A) (h2 : a ∉ A \ B)
  (h3 : a ∈ B → a ∈ C) : a ∈ C := by
  apply h3 _
  define at h2
  demorgan at h2; conditional at h2
  done
```

Tactic State in Infoview

```
U : Type
A B C : Set U
a : U
h1 : a ∈ A
h2 : a ∈ A → a ∈ B
h3 : a ∈ B → a ∈ C
↑ a ∈ B
```

And now it is easy to complete the proof by applying modus ponens, using `h2` and `h1`:

Lean File

```
theorem Like_Example_3_2_5
  (U : Type) (A B C : Set U) (a : U)
  (h1 : a ∈ A) (h2 : a ∉ A \ B)
  (h3 : a ∈ B → a ∈ C) : a ∈ C := by
  apply h3 _
  define at h2
  demorgan at h2; conditional at h2
  show a ∈ B from h2 h1
  done
```

Tactic State in Infoview

```
No goals
```

We will see many more uses of the `apply` tactic later in this book.

3.3. Proofs Involving Quantifiers

Sections 3.1 and 3.2 of *HTPI* contain several proofs that involve algebraic reasoning. Although one can do such proofs in Lean, it requires ideas that we are not ready to introduce yet. So for the moment we will stick to proofs involving only logic and set theory.

Exercises

Fill in proofs of the following theorems. All of them are based on exercises in *HTPI*. You can find these exercises in the file `Chap3Ex.lean` in the HTPI Lean package, and you can enter your solutions in that file.

1. `theorem Exercise_3_2_1a (P Q R : Prop)`

`(h1 : P → Q) (h2 : Q → R) : P → R := by`

[done](#)

2. `theorem Exercise_3_2_1b (P Q R : Prop)`

`(h1 : ¬R → (P → ¬Q)) : P → (Q → R) := by`

[done](#)

3. `theorem Exercise_3_2_2a (P Q R : Prop)`

`(h1 : P → Q) (h2 : R → ¬Q) : P → ¬R := by`

[done](#)

4. `theorem Exercise_3_2_2b (P Q : Prop)`

`(h1 : P) : Q → ¬(Q → ¬P) := by`

[done](#)

3.3. Proofs Involving Quantifiers

In the notation used in *HTPI*, if $P(x)$ is a statement about x , then $\forall x P(x)$ means “for all x , $P(x)$,” and $\exists x P(x)$ means “there exists at least one x such that $P(x)$.” The letter P here does not stand for a proposition; it is only when it is applied to some object x that we get a proposition. We will say that P is a *predicate*, and when we apply P to an object x we get the proposition $P(x)$. You might want to think of the predicate P as representing some property that an object might have, and the proposition $P(x)$ asserts that x has that property.

To use a predicate in Lean, you must tell Lean the type of objects to which it applies. If U is a type, then `Pred U` is the type of predicates that apply to objects of type U . If P has type `Pred`

3.3. Proofs Involving Quantifiers

U (that is, P is a predicate applying to objects of type U) and x has type U , then to apply P to x we just write $P\ x$ (with a space but no parentheses). Thus, if we have $P : \text{Pred } U$ and $x : U$, then $P\ x$ is an expression of type Prop . That is, $P\ x$ is a proposition, and its meaning is that x has the property represented by the predicate P .

There are a few differences between the way quantified statements are written in *HTPI* and the way they are written in Lean. First of all, when we apply a quantifier to a variable in Lean we will specify the type of the variable explicitly. Also, Lean requires that after specifying the variable and its type, you must put a comma before the proposition to which the quantifier is applied. Thus, if P has type $\text{Pred } U$, then to say that P holds for all objects of type U we would write $\forall (x : U), P\ x$. Similarly, $\exists (x : U), P\ x$ is the proposition asserting that there exists at least one x of type U such that $P\ x$.

And there is one more important difference between the way quantified statements are written in *HTPI* and Lean. In *HTPI*, a quantifier is interpreted as applying to as little as possible. Thus, $\forall x P(x) \wedge Q(x)$ is interpreted as $(\forall x P(x)) \wedge Q(x)$; if you want the quantifier $\forall x$ to apply to the entire statement $P(x) \wedge Q(x)$ you must use parentheses and write $\forall x(P(x) \wedge Q(x))$. The convention in Lean is exactly the opposite: a quantifier applies to as much as possible. Thus, Lean will interpret $\forall (x : U), P\ x \wedge Q\ x$ as meaning $\forall (x : U), (P\ x \wedge Q\ x)$. If you want the quantifier to apply to only $P\ x$, then you must use parentheses and write $(\forall (x : U), P\ x) \wedge Q\ x$.

With this preparation, we are ready to consider how to write proofs involving quantifiers in Lean. The most common way to prove a goal of the form $\forall (x : U), P\ x$ is to use the following strategy (*HTPI* p. 114):

To prove a goal of the form $\forall (x : U), P\ x$:

Let x stand for an arbitrary object of type U and prove $P\ x$. If the letter x is already being used in the proof to stand for something, then you must choose an unused variable, say y , to stand for the arbitrary object, and prove $P\ y$.

To do this in Lean, you should use the tactic `fix x : U`, which tells Lean to treat x as standing for some fixed but arbitrary object of type U . This has the following effect on the tactic state:

Tactic State Before Using Strategy

```
:  
↑ ⊢ ∀ (x : U), P x
```

Tactic State After Using Strategy

```
:  
x : U  
↑ ⊢ P x
```

To use a given of the form $\forall (x : U), P\ x$, we usually apply a rule of inference called *universal instantiation*, which is described by the following proof strategy (*HTPI* p. 121):

To use a given of the form $\forall (x : U), P x$:

You may plug in any value of type U , say a , for x and use this given to conclude that $P a$ is true.

This strategy says that if you have $h : \forall (x : U), P x$ and $a : U$, then you can infer $P a$. Indeed, in this situation Lean will recognize $h a$ as a proof of $P a$. For example, you can write `have h' : P a := h a` in a Lean tactic-mode proof, and Lean will add $h' : P a$ to the tactic state. Note that a here need not be simply a variable; it can be any expression denoting an object of type U .

Let's try these strategies out in a Lean proof. In Lean, if you don't want to give a theorem a name, you can simply call it an `example` rather than a `theorem`, and then there is no need to give it a name. In the following example, you can enter the symbol \forall by typing `\forallall` or `\forall`, and you can enter \exists by typing `\existsexists` or `\ex`.

Lean File

```
example (U : Type) (P Q : Pred U)
  (h1 : ∀ (x : U), P x → ¬Q x)
  (h2 : ∀ (x : U), Q x) :
  ¬∃ (x : U), P x := by
  done
```

Tactic State in Infoview

U : Type
P Q : Pred U
h1 : ∀ (x : U), P x → ¬Q x
h2 : ∀ (x : U), Q x
↑ ¬∃ (x : U), P x

To use the givens $h1$ and $h2$, we will probably want to use universal instantiation. But to do that we would need an object of type U to plug in for x in $h1$ and $h2$, and there is no object of type U in the tactic state. So at this point, we can't apply universal instantiation to $h1$ and $h2$. We should watch for an object of type U to come up in the course of the proof, and consider applying universal instantiation if one does. Until then, we turn our attention to the goal.

The goal is a negative statement, so we begin by reexpressing it as an equivalent positive statement, using a quantifier negation law. The tactic `quant_neg` applies a quantifier negation law to rewrite the goal. As with the other tactics for applying logical equivalences, you can write `quant_neg` at h if you want to apply a quantifier negation law to a given h . The effect of the tactic can be summarized as follows:

quant_neg Tactic		
$\neg\forall (x : U), P x$	is changed to	$\exists (x : U), \neg P x$
$\neg\exists (x : U), P x$	is changed to	$\forall (x : U), \neg P x$
$\forall (x : U), P x$	is changed to	$\neg\exists (x : U), \neg P x$
$\exists (x : U), P x$	is changed to	$\neg\forall (x : U), \neg P x$

Using the `quant_neg` tactic leads to the following result.

3.3. Proofs Involving Quantifiers

Lean File

```
example (U : Type) (P Q : Pred U)
  (h1 : ∀ (x : U), P x → ¬Q x)
  (h2 : ∀ (x : U), Q x) :
  ¬∃ (x : U), P x := by
quant_neg      --Goal is now ∀ (x : U), ¬P x
done
```

Tactic State in Infoview

U : Type
P Q : Pred U
h1 : ∀ (x : U), P x → ¬Q x
h2 : ∀ (x : U), Q x
↑ ∀ (x : U), ¬P x

Now the goal starts with \forall , so we use the strategy above and introduce an arbitrary object of type U . Since the variable x occurs as a bound variable in several statements in this theorem, it might be best to use a different letter for the arbitrary object; this isn't absolutely necessary, but it may help to avoid confusion. So our next tactic is `fix y : U`.

Lean File

```
example (U : Type) (P Q : Pred U)
  (h1 : ∀ (x : U), P x → ¬Q x)
  (h2 : ∀ (x : U), Q x) :
  ¬∃ (x : U), P x := by
quant_neg      --Goal is now ∀ (x : U), ¬P x
fix y : U
done
```

Tactic State in Infoview

U : Type
P Q : Pred U
h1 : ∀ (x : U), P x → ¬Q x
h2 : ∀ (x : U), Q x
y : U
↑ ¬P y

Now we have an object of type U in the tactic state, namely, y . So let's try applying universal instantiation to $h1$ and $h2$ and see if it helps.

Lean File

```
example (U : Type) (P Q : Pred U)
  (h1 : ∀ (x : U), P x → ¬Q x)
  (h2 : ∀ (x : U), Q x) :
  ¬∃ (x : U), P x := by
quant_neg      --Goal is now ∀ (x : U), ¬P x
fix y : U
have h3 : P y → ¬Q y := h1 y
have h4 : Q y := h2 y
done
```

Tactic State in Infoview

U : Type
P Q : Pred U
h1 : ∀ (x : U), P x → ¬Q x
h2 : ∀ (x : U), Q x
y : U
h3 : P y → ¬Q y
h4 : Q y
↑ ¬P y

We're almost done, because the goal now follows easily from $h3$ and $h4$. If we use the contrapositive law to rewrite $h3$ as $Q y \rightarrow \neg P y$, then we can apply modus ponens to the rewritten $h3$ and $h4$ to reach the goal:

3.3. Proofs Involving Quantifiers

Lean File

```
example (U : Type) (P Q : Pred U)
  (h1 : ∀ (x : U), P x → ¬Q x)
  (h2 : ∀ (x : U), Q x) :
  ¬∃ (x : U), P x := by
quant_neg      --Goal is now ∀ (x : U), ¬P x
fix y : U
have h3 : P y → ¬Q y := h1 y
have h4 : Q y := h2 y
contrapos at h3 --Now h3 : Q y → ¬P y
show ¬P y from h3 h4
done
```

Tactic State in Infoview

No goals

Our next example is a theorem of set theory. You already know how to type a few set theory symbols in Lean, but you'll need a few more for our next example. Here's a summary of the most important set theory symbols and how to type them in Lean.

Symbol	How To Type It
\in	\in
\notin	\notin or \inn
\subseteq	\sub
$\not\subseteq$	\subn
$=$	=
\neq	\ne
\cup	\union or \cup
\cap	\inter or \cap
\setminus	\setminus
Δ	\symmdiff
\emptyset	\emptysetset
\mathcal{P}	\powerset

With this preparation, we can turn to our next example.

Lean File

```
example (U : Type) (A B C : Set U) (h1 : A ⊆ B ∪ C)
  (h2 : ∀ (x : U), x ∈ A → x ∉ B) : A ⊆ C := by
done
```

Tactic State in Infoview

U : Type
A B C : Set U
h1 : A ⊆ B ∪ C
h2 : ∀ x ∈ A, x ∉ B
 $\vdash A \subseteq C$

Notice that in the Infoview, Lean has written h2 as $\forall x \in A, x \notin B$, using a bounded quantifier. As explained in Section 2.2 of *HTPI* (see p. 72), this is a shorter way of writing the statement

3.3. Proofs Involving Quantifiers

$\forall (x : U), x \in A \rightarrow x \notin B$. We begin by using the `define` tactic to write out the definition of the goal.

Lean File

```
example (U : Type) (A B C : Set U) (h1 : A ⊆ B ∪ C)
  (h2 : ∀ (x : U), x ∈ A → x ∉ B) : A ⊆ C := by
  define --Goal : ∀ {a : U}, a ∈ A → a ∈ C
  done
```

Tactic State in Infoview

U : Type
A B C : Set U
h1 : A ⊆ B ∪ C
h2 : ∀ x ∈ A, x ∉ B
$\vdash \forall \{a : U\},$
a ∈ A → a ∈ C

Notice that Lean's definition of the goal starts with $\forall \{a : U\}$, not $\forall (a : U)$. Why did Lean use those funny double braces rather than parentheses? We'll return to that question shortly. The difference doesn't affect our next steps, which are to introduce an arbitrary object y of type U and assume $y \in A$.

Lean File

```
example (U : Type) (A B C : Set U) (h1 : A ⊆ B ∪ C)
  (h2 : ∀ (x : U), x ∈ A → x ∉ B) : A ⊆ C := by
  define --Goal : ∀ {a : U}, a ∈ A → a ∈ C
  fix y : U
  assume h3 : y ∈ A
  done
```

Tactic State in Infoview

U : Type
A B C : Set U
h1 : A ⊆ B ∪ C
h2 : ∀ x ∈ A, x ∉ B
y : U
h3 : y ∈ A
$\vdash y \in C$

Now we can combine $h2$ and $h3$ to conclude that $y \notin B$. Since we have $y : U$, by universal instantiation, $h2 y$ is a proof of $y \in A \rightarrow y \notin B$, and therefore by modus ponens, $h2 y h3$ is a proof of $y \notin B$.

Lean File

```
example (U : Type) (A B C : Set U) (h1 : A ⊆ B ∪ C)
  (h2 : ∀ (x : U), x ∈ A → x ∉ B) : A ⊆ C := by
  define --Goal : ∀ {a : U}, a ∈ A → a ∈ C
  fix y : U
  assume h3 : y ∈ A
  have h4 : y ∉ B := h2 y h3
  done
```

Tactic State in Infoview

U : Type
A B C : Set U
h1 : A ⊆ B ∪ C
h2 : ∀ x ∈ A, x ∉ B
y : U
h3 : y ∈ A
h4 : y ∉ B
$\vdash y \in C$

We should be able to use similar reasoning to combine $h1$ and $h3$, if we first write out the definition of $h1$.

Lean File

```
example (U : Type) (A B C : Set U) (h1 : A ⊆ B ∪ C)
  (h2 : ∀ (x : U), x ∈ A → x ∉ B) : A ⊆ C := by
  define --Goal : ∀ {a : U}, a ∈ A → a ∈ C
  fix y : U
  assume h3 : y ∈ A
  have h4 : y ∉ B := h2 y h3
  define at h1 --h1 : ∀ {a : U}, a ∈ A → a ∈ B ∪ C
  done
```

Tactic State in Infoview

```
U : Type
A B C : Set U
h1 : ∀ {a : U},
  a ∈ A → a ∈ B ∪ C
h2 : ∀ x ∈ A, x ∉ B
y : U
h3 : y ∈ A
h4 : y ∉ B
↑ y ∈ C
```

Once again, Lean has used double braces to define `h1`, and now we are ready to explain what they mean. If the definition had been `h1 : ∀ (a : U), a ∈ A → a ∈ B ∪ C`, then exactly as in the previous step, `h1 y h3` would be a proof of `y ∈ B ∪ C`. The use of double braces in the definition `h1 : ∀ {a : U}, a ∈ A → a ∈ B ∪ C` means that you don't need to tell Lean that `y` is being plugged in for `a` in the universal instantiation step; Lean will figure that out on its own. Thus, you can just write `h1 h3` as a proof of `y ∈ B ∪ C`. Indeed, if you write `h1 y h3` then you will get an error message, because Lean expects *not* to be told what to plug in for `a`. You might think of the definition of `h1` as meaning `h1 : _ ∈ A → _ ∈ B ∪ C`, where the blanks can be filled in with anything of type `U` (with the same thing being put in both blanks). When you ask Lean to apply modus ponens by combining this statement with `h3 : y ∈ A`, Lean figures out that in order for modus ponens to apply, the blanks must be filled in with `y`.

In this situation, the `a` in `h1` is called an *implicit argument*. What this means is that, when `h1` is applied to make an inference in a proof, the value to be assigned to `a` is not specified explicitly; rather, the value is inferred by Lean. We will see many more examples of implicit arguments later in this book. In fact, there are two slightly different kinds of implicit arguments in Lean. One kind is indicated using the double braces `{ }` used in this example, and the other is indicated using curly braces, `{ }`. The difference between these two kinds of implicit arguments won't be important in this book; all that will matter to us is that if you see either `∀ {a : U}` or `∀ {a : U}` rather than `∀ (a : U)`, then you must remember that `a` is an implicit argument.

Lean File

```
example (U : Type) (A B C : Set U) (h1 : A ⊆ B ∪ C)
  (h2 : ∀ (x : U), x ∈ A → x ∉ B) : A ⊆ C := by
  define --Goal : ∀ {a : U}, a ∈ A → a ∈ C
  fix y : U
  assume h3 : y ∈ A
  have h4 : y ∉ B := h2 y h3
  define at h1 --h1 : ∀ {a : U}, a ∈ A → a ∈ B ∪ C
  have h5 : y ∈ B ∪ C := h1 h3
  done
```

Tactic State in Infoview

```
U : Type
A B C : Set U
h1 : ∀ {a : U},
  a ∈ A → a ∈ B ∪ C
h2 : ∀ x ∈ A, x ∉ B
y : U
h3 : y ∈ A
h4 : y ∉ B
h5 : y ∈ B ∪ C
↑ y ∈ C
```

If Lean was able to figure out that y should be plugged in for a in $h1$ in this step, couldn't it have figured out that y should be plugged in for x in $h2$ in the previous `have` step? The answer is yes. Of course, in $h2$, x was not an implicit argument, so Lean wouldn't *automatically* figure out what to plug in for x . But we could have asked it to figure it out by writing the proof in the previous step as $h2 _ h3$ rather than $h2 \ y \ h3$. In a term-mode proof, an underscore represents a blank to be filled in by Lean. Try changing the earlier step of the proof to `have h4 : y ∉ B := h2 _ h3` and you will see that Lean will accept it. Of course, in this case this doesn't save us any typing, but in some situations it is useful to let Lean figure out some part of a proof.

Lean's ability to fill in blanks in term-mode proofs is limited. For example, if you try changing the previous step to `have h4 : y ∉ B := h2 y _`, you'll get a red squiggle under the blank, and the error message in the Infoview pane will say `don't know how to synthesize placeholder`. In other words, Lean was unable to figure out how to fill in the blank in this case. In future proofs you might try replacing some expressions with blanks to get a feel for what Lean can and cannot figure out for itself.

Continuing with the proof, we see that we're almost done, because we can combine $h4$ and $h5$ to reach our goal. To see how, we first write out the definition of $h5$.

Lean File

```
example (U : Type) (A B C : Set U) (h1 : A ⊆ B ∪ C)
      (h2 : ∀ (x : U), x ∈ A → x ∉ B) : A ⊆ C := by
  define --Goal : ∀ {a : U}, a ∈ A → a ∈ C
  fix y : U
  assume h3 : y ∈ A
  have h4 : y ∉ B := h2 y h3
  define at h1 --h1 : ∀ {a : U}, a ∈ A → a ∈ B ∪ C
  have h5 : y ∈ B ∪ C := h1 h3
  define at h5 --h5 : y ∈ B ∨ y ∈ C
  done
```

Tactic State in Infoview

$\text{U} : \text{Type}$
$\text{A } \text{B } \text{C} : \text{Set } \text{U}$
$\text{h1} : \forall \{a : \text{U}\}, a \in \text{A} \rightarrow a \in \text{B} \cup \text{C}$
$\text{h2} : \forall x \in \text{A}, x \notin \text{B}$
$\text{y} : \text{U}$
$\text{h3} : y \in \text{A}$
$\text{h4} : y \notin \text{B}$
$\text{h5} : y \in \text{B} \vee y \in \text{C}$
$\vdash y \in \text{C}$

A conditional law will convert $h5$ to $y \notin B \rightarrow y \in C$, and then modus ponens with $h4$ will complete the proof.

3.3. Proofs Involving Quantifiers

Lean File

```
example (U : Type) (A B C : Set U) (h1 : A ⊆ B ∪ C)
  (h2 : ∀ (x : U), x ∈ A → x ∉ B) : A ⊆ C := by
  define --Goal : ∀ {a : U}, a ∈ A → a ∈ C
  fix y : U
  assume h3 : y ∈ A
  have h4 : y ∉ B := h2 y h3
  define at h1 --h1 : ∀ {a : U}, a ∈ A → a ∈ B ∪ C
  have h5 : y ∈ B ∪ C := h1 h3
  define at h5 --h5 : y ∈ B ∨ y ∈ C
  conditional at h5 --h5 : y ∉ B → y ∈ C
  show y ∈ C from h5 h4
  done
```

Tactic State in Infoview

No goals

Next we turn to strategies for working with existential quantifiers (*HTPI* p. 118).

To prove a goal of the form $\exists (x : U), P x$:

Find a value of x , say a , for which you think $P a$ is true, and prove $P a$.

This strategy is based on the fact that if you have $a : U$ and $h : P a$, then you can infer $\exists (x : U), P x$. Indeed, in this situation the expression `Exists.intro a h` is a Lean term-mode proof of $\exists (x : U), P x$. The name `Exists.intro` indicates that this is a rule for introducing an existential quantifier.

Note that, as with the universal instantiation rule, a here can be any expression denoting an object of type U ; it need not be simply a variable. For example, if A and B have type `Set U`, F has type `Set (Set U)`, and you have a given $h : A ∪ B ∈ F$, then `Exists.intro (A ∪ B) h` is a proof of $\exists (x : Set U), x ∈ F$.

As suggested by the strategy above, we will often want to use the `Exists.intro` rule in situations in which our goal is $\exists (x : U), P x$ and we have an object a of type U that we think makes $P a$ true, but we don't yet have a proof of $P a$. In that situation we can use the tactic `apply Exists.intro a _`. Recall that the `apply` tactic asks Lean to figure out what to put in the blank to turn `Exists.intro a _` into a proof of the goal. Lean will figure out that what needs to go in the blank is a proof of $P a$, so it sets $P a$ to be the goal. In other words, the tactic `apply Exists.intro a _` has the following effect on the tactic state:

Tactic State Before Using Strategy

```
:  
a : U  
↑ ∃ (x : U), P x
```

Tactic State After Using Strategy

```
:  
a : U  
↑ P a
```

3.3. Proofs Involving Quantifiers

Our strategy for using an existential given is a rule that is called *existential instantiation* in *HTPI* (*HTPI* p. 120):

To use a given of the form $\exists (x : U), P x$:

Introduce a new variable, say u , into the proof to stand for an object of type U for which $P u$ is true.

Suppose that, in a Lean proof, you have $h : \exists (x : U), P x$. To apply the existential instantiation rule, you would use the tactic `obtain (u : U) (h' : P u)` from h . This tactic introduces into the tactic state both a new variable u of type U and also the identifier h' for the new given $P u$. Note that h can be any proof of a statement of the form $\exists (x : U), P x$; it need not be just a single identifier.

Often, if your goal is an existential statement $\exists (x : U), P x$, you won't be able to use the strategy above for existential goals right away, because you won't know what object a to use in the tactic `apply Exists.intro a _`. You may have to wait until a likely candidate for a pops up in the course of the proof. On the other hand, it is usually best to use the `obtain` tactic right away if you have an existential given. This is illustrated in our next example.

Lean File

```
example (U : Type) (P Q : Pred U)
  (h1 : ∀ (x : U), ∃ (y : U), P x → ¬ Q y)
  (h2 : ∃ (x : U), ∀ (y : U), P x → Q y) :
  ∃ (x : U), ¬P x := by
  done
```

Tactic State in Infoview

U : Type
P Q : Pred U
h1 : ∀ (x : U), ∃ (y : U), P x → ¬Q y
h2 : ∃ (x : U), ∀ (y : U), P x → Q y
↑ $\exists (x : U), \neg P x$

The goal is the existential statement $\exists (x : U), \neg P x$, and our strategy for existential goals says that we should try to find an object a of type U that we think would make the statement $\neg P a$ true. But we don't have any objects of type U in the tactic state, so it looks like we can't use that strategy yet. Similarly, we can't use the given $h1$ yet, since we have nothing to plug in for x in $h1$. However, $h2$ is an existential given, and we can use it right away.

3.3. Proofs Involving Quantifiers

Lean File

```
example (U : Type) (P Q : Pred U)
  (h1 : ∀ (x : U), ∃ (y : U), P x → ¬ Q y)
  (h2 : ∃ (x : U), ∀ (y : U), P x → Q y) :
  ∃ (x : U), ¬P x := by
obtain (a : U)
  (h3 : ∀ (y : U), P a → Q y) from h2
done
```

Tactic State in Infoview

```
U : Type
P Q : Pred U
h1 : ∀ (x : U), ∃ (y : U),
  P x → ¬Q y
h2 : ∃ (x : U), ∀ (y : U),
  P x → Q y
a : U
h3 : ∀ (y : U), P a → Q y
↑ ∃ (x : U), ¬P x
```

Now that we have `a : U`, we can apply universal instantiation to `h1`, plugging in `a` for `x`.

Lean File

```
example (U : Type) (P Q : Pred U)
  (h1 : ∀ (x : U), ∃ (y : U), P x → ¬ Q y)
  (h2 : ∃ (x : U), ∀ (y : U), P x → Q y) :
  ∃ (x : U), ¬P x := by
obtain (a : U)
  (h3 : ∀ (y : U), P a → Q y) from h2
have h4 : ∃ (y : U), P a → ¬ Q y := h1 a
done
```

Tactic State in Infoview

```
U : Type
P Q : Pred U
h1 : ∀ (x : U), ∃ (y : U),
  P x → ¬Q y
h2 : ∃ (x : U), ∀ (y : U),
  P x → Q y
a : U
h3 : ∀ (y : U), P a → Q y
h4 : ∃ (y : U), P a → ¬Q y
↑ ∃ (x : U), ¬P x
```

By the way, this is another case in which Lean could have figured out a part of the proof on its own. Try changing `h1 a` in the last step to `h1 _`, and you'll see that Lean will be able to figure out how to fill in the blank.

Our new given `h4` is another existential statement, so again we use it right away to introduce another object of type `U`. Since this object might not be the same as `a`, we must give it a different name. (Indeed, if you try to use the name `a` again, Lean will give you an error message.)

Lean File

```

example (U : Type) (P Q : Pred U)
  (h1 : ∀ (x : U), ∃ (y : U), P x → ¬ Q y)
  (h2 : ∃ (x : U), ∀ (y : U), P x → Q y) :
  ∃ (x : U), ¬P x := by
obtain (a : U)
  (h3 : ∀ (y : U), P a → Q y) from h2
have h4 : ∃ (y : U), P a → ¬ Q y := h1 a
obtain (b : U) (h5 : P a → ¬ Q b) from h4
done

```

Tactic State in Infoview

```

U : Type
P Q : Pred U
h1 : ∀ (x : U), ∃ (y : U),
  P x → ¬Q y
h2 : ∃ (x : U), ∀ (y : U),
  P x → Q y
a : U
h3 : ∀ (y : U), P a → Q y
h4 : ∃ (y : U), P a → ¬Q y
b : U
h5 : P a → ¬Q b
↑ ∃ (x : U), ¬P x

```

We have not yet used `h3`. We could plug in either `a` or `b` for `y` in `h3`, but a little thought should show you that plugging in `b` is more useful.

Lean File

```

example (U : Type) (P Q : Pred U)
  (h1 : ∀ (x : U), ∃ (y : U), P x → ¬ Q y)
  (h2 : ∃ (x : U), ∀ (y : U), P x → Q y) :
  ∃ (x : U), ¬P x := by
obtain (a : U)
  (h3 : ∀ (y : U), P a → Q y) from h2
have h4 : ∃ (y : U), P a → ¬ Q y := h1 a
obtain (b : U) (h5 : P a → ¬ Q b) from h4
have h6 : P a → Q b := h3 b
done

```

Tactic State in Infoview

```

U : Type
P Q : Pred U
h1 : ∀ (x : U), ∃ (y : U),
  P x → ¬Q y
h2 : ∃ (x : U), ∀ (y : U),
  P x → Q y
a : U
h3 : ∀ (y : U), P a → Q y
h4 : ∃ (y : U), P a → ¬Q y
b : U
h5 : P a → ¬Q b
h6 : P a → Q b
↑ ∃ (x : U), ¬P x

```

Now look at `h5` and `h6`. They show that `P a` leads to contradictory conclusions, `¬Q b` and `Q b`. This means that `P a` must be false. We finally know what value of `x` to use to prove the goal.

3.3. Proofs Involving Quantifiers

Lean File

```
example (U : Type) (P Q : Pred U)
  (h1 : ∀ (x : U), ∃ (y : U), P x → ¬ Q y)
  (h2 : ∃ (x : U), ∀ (y : U), P x → Q y) :
  ∃ (x : U), ¬P x := by
obtain (a : U)
  (h3 : ∀ (y : U), P a → Q y) from h2
have h4 : ∃ (y : U), P a → ¬ Q y := h1 a
obtain (b : U) (h5 : P a → ¬ Q b) from h4
have h6 : P a → Q b := h3 b
apply Exists.intro a _
done
```

Tactic State in Infoview

```
U : Type
P Q : Pred U
h1 : ∀ (x : U), ∃ (y : U),
  P x → ¬Q y
h2 : ∃ (x : U), ∀ (y : U),
  P x → Q y
a : U
h3 : ∀ (y : U), P a → Q y
h4 : ∃ (y : U), P a → ¬Q y
b : U
h5 : P a → ¬Q b
h6 : P a → Q b
↑ ¬P a
```

Since the goal is now a negative statement that cannot be reexpressed as a positive statement, we use proof by contradiction.

Lean File

```
example (U : Type) (P Q : Pred U)
  (h1 : ∀ (x : U), ∃ (y : U), P x → ¬ Q y)
  (h2 : ∃ (x : U), ∀ (y : U), P x → Q y) :
  ∃ (x : U), ¬P x := by
obtain (a : U)
  (h3 : ∀ (y : U), P a → Q y) from h2
have h4 : ∃ (y : U), P a → ¬ Q y := h1 a
obtain (b : U) (h5 : P a → ¬ Q b) from h4
have h6 : P a → Q b := h3 b
apply Exists.intro a _
by_contra h7
done
```

Tactic State in Infoview

```
U : Type
P Q : Pred U
h1 : ∀ (x : U), ∃ (y : U),
  P x → ¬Q y
h2 : ∃ (x : U), ∀ (y : U),
  P x → Q y
a : U
h3 : ∀ (y : U), P a → Q y
h4 : ∃ (y : U), P a → ¬Q y
b : U
h5 : P a → ¬Q b
h6 : P a → Q b
h7 : P a
↑ False
```

Now h5 h7 is a proof of $\neg Q b$ and h6 h7 is a proof of $Q b$, so h5 h7 (h6 h7) is a proof of False.

3.3. Proofs Involving Quantifiers

Lean File

```
example (U : Type) (P Q : Pred U)
  (h1 : ∀ (x : U), ∃ (y : U), P x → ¬ Q y)
  (h2 : ∃ (x : U), ∀ (y : U), P x → Q y) :
  ∃ (x : U), ¬P x := by
obtain (a : U)
  (h3 : ∀ (y : U), P a → Q y) from h2
have h4 : ∃ (y : U), P a → ¬ Q y := h1 a
obtain (b : U) (h5 : P a → ¬ Q b) from h4
have h6 : P a → Q b := h3 b
apply Exists.intro a _
by_contra h7
show False from h5 h7 (h6 h7)
done
```

Tactic State in Infoview

No goals

We conclude this section with the theorem from Example 3.3.5 in *HTPI*. That theorem concerns a union of a family of sets. In *HTPI*, such a union is written using a large union symbol, \bigcup . Lean uses the symbol \cup_0 , which is entered by typing `\U0` (that is, backslash–capital U–zero). For an intersection of a family of sets, Lean uses \cap_0 , typed as `\I0`.

Lean File

```
theorem Example_3_3_5 (U : Type) (B : Set U)
  (F : Set (Set U)) : U₀ F ⊆ B → F ⊆ ℙ B := by
done
```

Tactic State in Infoview

U : Type
B : Set U
F : Set (Set U)
 $\vdash U_0 F \subseteq B \rightarrow F \subseteq \mathcal{P} B$

Note that F has type `Set (Set U)`, which means that it is a set whose elements are sets of objects of type U . Since the goal is a conditional statement, we assume the antecedent and set the consequent as our goal. We'll also write out the definition of the new goal.

Lean File

```
theorem Example_3_3_5 (U : Type) (B : Set U)
  (F : Set (Set U)) : U₀ F ⊆ B → F ⊆ ℙ B := by
assume h1 : U₀ F ⊆ B
define
done
```

Tactic State in Infoview

U : Type
B : Set U
F : Set (Set U)
h1 : $U_0 F \subseteq B$
 $\vdash \forall \{a : Set U\},$
 $a \in F \rightarrow a \in \mathcal{P} B$

Based on the form of the goal, we introduce an arbitrary object x of type `Set U` and assume $x \in F$. The new goal will be $x \in \mathcal{P} B$. The `define` tactic works out that this means $x \subseteq B$, which can be further expanded to $\forall \{a : U\}, a \in x \rightarrow a \in B$.

3.3. Proofs Involving Quantifiers

Lean File

```
theorem Example_3_3_5 (U : Type) (B : Set U)
  (F : Set (Set U)) : U₀ F ⊆ B → F ⊆ ℙ B := by
  assume h1 : U₀ F ⊆ B
  define
  fix x : Set U
  assume h2 : x ∈ F
  define
  done
```

Tactic State in Infoview

```
U : Type
B : Set U
F : Set (Set U)
h1 : U₀ F ⊆ B
x : Set U
h2 : x ∈ F
↑ ∀ {a : U},
  a ∈ x → a ∈ B
```

Once again the form of the goal dictates our next steps: introduce an arbitrary y of type U and assume $y \in x$.

Lean File

```
theorem Example_3_3_5 (U : Type) (B : Set U)
  (F : Set (Set U)) : U₀ F ⊆ B → F ⊆ ℙ B := by
  assume h1 : U₀ F ⊆ B
  define
  fix x : Set U
  assume h2 : x ∈ F
  define
  fix y : U
  assume h3 : y ∈ x
  done
```

Tactic State in Infoview

```
U : Type
B : Set U
F : Set (Set U)
h1 : U₀ F ⊆ B
x : Set U
h2 : x ∈ F
y : U
h3 : y ∈ x
↑ y ∈ B
```

The goal can be analyzed no further, so we turn to the givens. We haven't used $h1$ yet. To see how to use it, we write out its definition.

Lean File

```
theorem Example_3_3_5 (U : Type) (B : Set U)
  (F : Set (Set U)) : U₀ F ⊆ B → F ⊆ ℙ B := by
  assume h1 : U₀ F ⊆ B
  define
  fix x : Set U
  assume h2 : x ∈ F
  define
  fix y : U
  assume h3 : y ∈ x
  define at h1
  done
```

Tactic State in Infoview

```
U : Type
B : Set U
F : Set (Set U)
h1 : ∀ {a : U},
  a ∈ U₀ F → a ∈ B
x : Set U
h2 : x ∈ F
y : U
h3 : y ∈ x
↑ y ∈ B
```

3.3. Proofs Involving Quantifiers

Now we see that we can try to use `h1 _` to reach our goal. Indeed, `h1 _` would be a proof of the goal if we could fill in the blank with a proof of $y \in U_0 F$. So we use the `apply h1 _` tactic.

Lean File

```
theorem Example_3_3_5 (U : Type) (B : Set U)
  (F : Set (Set U)) : U₀ F ⊆ B → F ⊆ ℙ B := by
  assume h1 : U₀ F ⊆ B
  define
  fix x : Set U
  assume h2 : x ∈ F
  define
  fix y : U
  assume h3 : y ∈ x
  define at h1
  apply h1 _
  done
```

Tactic State in Infoview

U : Type
B : Set U
F : Set (Set U)
h1 : $\forall \{a : U\},$
$a \in U_0 F \rightarrow a \in B$
x : Set U
h2 : $x \in F$
y : U
h3 : $y \in x$
$\vdash y \in U_0 F$

Once again we have a goal that can be analyzed by using the `define` tactic.

Lean File

```
theorem Example_3_3_5 (U : Type) (B : Set U)
  (F : Set (Set U)) : U₀ F ⊆ B → F ⊆ ℙ B := by
  assume h1 : U₀ F ⊆ B
  define
  fix x : Set U
  assume h2 : x ∈ F
  define
  fix y : U
  assume h3 : y ∈ x
  define at h1
  apply h1 _
  define
  done
```

Tactic State in Infoview

U : Type
B : Set U
F : Set (Set U)
h1 : $\forall \{a : U\},$
$a \in U_0 F \rightarrow a \in B$
x : Set U
h2 : $x \in F$
y : U
h3 : $y \in x$
$\vdash \exists t \in F, y \in t$

Our goal now is $\exists (t : \text{Set } U), t \in F \wedge y \in t$, although once again Lean has used a bounded quantifier to write this in a shorter form. So we look for a value of t that will make the statement $t \in F \wedge y \in t$ true. The givens `h2` and `h3` tell us that x is such a value, so as described earlier our next tactic should be `apply Exists.intro x _`.

3.3. Proofs Involving Quantifiers

Lean File

```
theorem Example_3_3_5 (U : Type) (B : Set U)
  (F : Set (Set U)) : U₀ F ⊆ B → F ⊆ ℙ B := by
  assume h1 : U₀ F ⊆ B
  define
  fix x : Set U
  assume h2 : x ∈ F
  define
  fix y : U
  assume h3 : y ∈ x
  define at h1
  apply h1 _
  define
  apply Exists.intro x _
  done
```

Tactic State in Infoview

U	: Type
B	: Set U
F	: Set (Set U)
h1	: ∀ {a : U}, a ∈ U₀ F → a ∈ B
x	: Set U
h2	: x ∈ F
y	: U
h3	: y ∈ x
	↑ x ∈ F ∧ y ∈ x

Clearly the goal now follows from h2 and h3, but how do we write the proof in Lean? Since we need to introduce the “and” symbol \wedge , you shouldn’t be surprised to learn that the rule we need is called `And.intro`. Proof strategies for statements involving “and” will be the subject of the next section.

Lean File

```
theorem Example_3_3_5 (U : Type) (B : Set U)
  (F : Set (Set U)) : U₀ F ⊆ B → F ⊆ ℙ B := by
  assume h1 : U₀ F ⊆ B
  define
  fix x : Set U
  assume h2 : x ∈ F
  define
  fix y : U
  assume h3 : y ∈ x
  define at h1
  apply h1 _
  define
  apply Exists.intro x _
  show x ∈ F ∧ y ∈ x from And.intro h2 h3
  done
```

Tactic State in Infoview

No goals

You might want to compare the Lean proof above to the way the proof was written in *HTPI*. Here are the theorem and proof from *HTPI* (*HTPI* p. 125):

Theorem. Suppose B is a set and \mathcal{F} is a family of sets. If $\bigcup \mathcal{F} \subseteq B$ then $\mathcal{F} \subseteq \mathcal{P}(B)$.

3.4. Proofs Involving Conjunctions and Biconditionals

Proof. Suppose $\bigcup \mathcal{F} \subseteq B$. Let x be an arbitrary element of \mathcal{F} . Let y be an arbitrary element of x . Since $y \in x$ and $x \in \mathcal{F}$, by the definition of $\bigcup \mathcal{F}$, $y \in \bigcup \mathcal{F}$. But then since $\bigcup \mathcal{F} \subseteq B$, $y \in B$. Since y was an arbitrary element of x , we can conclude that $x \subseteq B$, so $x \in \mathcal{P}(B)$. But x was an arbitrary element of \mathcal{F} , so this shows that $\mathcal{F} \subseteq \mathcal{P}(B)$, as required. \square

Exercises

1. `theorem Exercise_3_3_1 (U : Type) (P Q : Pred U) (h1 : ∃ (x : U), P x → Q x) : (∀ (x : U), P x) → ∃ (x : U), Q x := by`
done
2. `theorem Exercise_3_3_8 (U : Type) (F : Set (Set U)) (A : Set U) (h1 : A ∈ F) : A ⊆ U₀ F := by`
done
3. `theorem Exercise_3_3_9 (U : Type) (F : Set (Set U)) (A : Set U) (h1 : A ∈ F) : ∩₀ F ⊆ A := by`
done
4. `theorem Exercise_3_3_10 (U : Type) (B : Set U) (F : Set (Set U)) (h1 : ∀ (A : Set U), A ∈ F → B ⊆ A) : B ⊆ ∩₀ F := by`
done
5. `theorem Exercise_3_3_13 (U : Type) (F G : Set (Set U)) : F ⊆ G → ∩₀ G ⊆ ∩₀ F := by`
done

3.4. Proofs Involving Conjunctions and Biconditionals

The strategies in *HTPI* for working with conjunctions are very simple (*HTPI* p. 130).

To prove a goal of the form $P \wedge Q$:

Prove P and Q separately.

We already saw an example, at the end of the last section, of the use of the rule `And.intro` to prove a conjunction. In general, if you have $h1 : P$ and $h2 : Q$, then `And.intro h1 h2` is a proof of $P \wedge Q$. It follows that if your goal is $P \wedge Q$ but you don't yet have proofs of P and Q , then you can use the tactic `apply And.intro _ _`. Lean will figure out that the blanks need to be filled in with proofs of P and Q , so it will ask you to prove P and Q separately, as suggested by the strategy above.

If you already have a proof of either P or Q , then you can fill in one of the blanks in the `apply` tactic. For example, if you have $h : P$, then you can write `apply And.intro h _`, and Lean will tell you that you just have to prove Q to complete the proof. Similarly, if you have $h : Q$, then `apply And.intro _ h` will lead to just the single goal P . There is also a shortcut you can use with the `apply` tactic: any blanks that come at the end of the tactic can be left out. So instead of `apply And.intro _ _`, you can just write `apply And.intro`, and instead of `apply And.intro h _`, you can write `apply And.intro h`. On the other hand, `apply And.intro _ h` can't be shortened; it is only blanks *at the end* that can be left out.

The strategy for a given that is a conjunction is similar (*HTPI* p. 131).

To use a given of the form $P \wedge Q$:

Treat this as two separate givens: P , and Q .

If you have a given $h : P \wedge Q$, then Lean will recognize $h.left$ as a proof of P , and $h.right$ as a proof of Q .

Here's an example that illustrates these strategies. It is similar to Example 3.4.1 in *HTPI*

Lean File

```
theorem Like_Example_3_4_1 (U : Type)
  (A B C D : Set U) (h1 : A ⊆ B)
  (h2 : ∃ (c : U), c ∈ C ∩ D) :
  A ∩ C ⊆ B \ D := by
  done
```

Tactic State in Infoview

$\text{U} : \text{Type}$ $\text{A B C D} : \text{Set U}$ $\text{h1} : A \subseteq B$ $\text{h2} : \exists (c : U), c \in C \cap D$ $\vdash A \cap C \subseteq B \setminus D$
--

The `define` tactic will rewrite the goal as $\forall \{a : U\}, a \in A \cap C \rightarrow a \in B \setminus D$, and then we can introduce an arbitrary $x : U$ and assume $x \in A \cap C$.

3.4. Proofs Involving Conjunctions and Biconditionals

Lean File

```
theorem Like_Example_3_4_1 (U : Type)
  (A B C D : Set U) (h1 : A ⊆ B)
  (h2 : ¬∃ (c : U), c ∈ C ∩ D) :
  A ∩ C ⊆ B \ D := by
  define
  fix x : U
  assume h3 : x ∈ A ∩ C
  done
```

Tactic State in Infoview

```
U : Type
A B C D : Set U
h1 : A ⊆ B
h2 : ¬∃ (c : U), c ∈ C ∩ D
x : U
h3 : x ∈ A ∩ C
↑ x ∈ B \ D
```

Now let's take a look at the definitions of `h3` and the goal:

Lean File

```
theorem Like_Example_3_4_1 (U : Type)
  (A B C D : Set U) (h1 : A ⊆ B)
  (h2 : ¬∃ (c : U), c ∈ C ∩ D) :
  A ∩ C ⊆ B \ D := by
  define
  fix x : U
  assume h3 : x ∈ A ∩ C
  define at h3; define
  done
```

Tactic State in Infoview

```
U : Type
A B C D : Set U
h1 : A ⊆ B
h2 : ¬∃ (c : U), c ∈ C ∩ D
x : U
h3 : x ∈ A ∧ x ∈ C
↑ x ∈ B ∧ x ∉ D
```

Since the goal is now a conjunction, we apply the strategy above by using the tactic `apply And.intro`.

Lean File

```
theorem Like_Example_3_4_1 (U : Type)
  (A B C D : Set U) (h1 : A ⊆ B)
  (h2 : ¬∃ (c : U), c ∈ C ∩ D) :
  A ∩ C ⊆ B \ D := by
  define
  fix x : U
  assume h3 : x ∈ A ∩ C
  define at h3; define
  apply And.intro
  done
```

Tactic State in Infoview

```
▼ case left
U : Type
A B C D : Set U
h1 : A ⊆ B
h2 : ¬∃ (c : U), c ∈ C ∩ D
x : U
h3 : x ∈ A ∧ x ∈ C
↑ x ∈ B
▼ case right
U : Type
A B C D : Set U
h1 : A ⊆ B
h2 : ¬∃ (c : U), c ∈ C ∩ D
x : U
h3 : x ∈ A ∧ x ∈ C
↑ x ∉ D
```

Look carefully at the tactic state. Lean has listed *two* goals, one after the other, and it has helpfully labeled them “case left” and “case right,” indicating that the first goal is to prove the left side of the conjunction and the second is to prove the right. The given information in both cases is the same, but in the first case the goal is $x \in B$, and in the second it is $x \notin D$. At this point, if we simply continue with the proof, Lean will interpret our tactics as applying to the first goal, until we achieve that goal. Once we achieve it, Lean will move on to the second goal.

However, we can make our proof more readable by separating and labeling the proofs of the two goals. To do this, we type a bullet (which looks like this: `.`) and then a comment describing the first goal. (To type a bullet, type `\.`—that is, backslash-period.) The proof of the first goal will appear below this line, indented further and ending with `done`. To prepare for this, we leave a blank line, type `tab` to increase the indenting, and then type `done`. Then we do the same for the second goal: on the next line, we return to the previous level of indenting and type a bullet and a comment describing the second goal. We follow this with a blank line and then an indented `done` to indicate the end of the proof of the second goal. We’re going to work on the first goal first, so we click on the first blank line to position the cursor there. The screen now looks like this:

Lean File

```

theorem Like_Example_3_4_1 (U : Type)
  (A B C D : Set U) (h1 : A ⊆ B)
  (h2 : ¬∃ (c : U), c ∈ C ∩ D) :
  A ∩ C ⊆ B \ D := by
  define
  fix x : U
  assume h3 : x ∈ A ∩ C
  define at h3; define
  apply And.intro
  · -- Proof that x ∈ B

  done
  · -- Proof that x ∉ D

  done
done

```

Tactic State in Infoview

```

▼ case left
U : Type
A B C D : Set U
h1 : A ⊆ B
h2 : ¬∃ (c : U), c ∈ C ∩ D
x : U
h3 : x ∈ A ∧ x ∈ C
↑ x ∈ B

```

Of course, there are red squiggles under both new occurrences of `done`, since neither goal has yet been achieved. We can work on the goals in either order by positioning the cursor on either blank line, and the Infoview pane will show the tactic state for the goal at the position of the cursor. In the display above, we have positioned the cursor on the first blank line, so the Infoview shows the tactic state for the first goal.

This first goal is easy: We have `h1 : A ⊆ B` and, as explained above, `h3.left : x ∈ A`. As we have seen in several previous examples, the tactic `define at h1` will rewrite `h1` as `∀ {a : U}, a ∈ A → a ∈ B`, and then `h1 h3.left` will be a proof of `x ∈ B`. And now we'll let you in on a little secret: usually the `define` tactic isn't really necessary. You may find the `define` tactic to be useful in many situations, because it helps you see what a statement means. But *Lean* doesn't need to be told to work out what the statement means; it will do that automatically. So we can skip the `define` tactic and just give `h1 h3.left` as a proof of `x ∈ B`. In general, if you have `h1 : A ⊆ B` and `h2 : x ∈ A`, then *Lean* will recognize `h1 h2` as a proof of `x ∈ B`. Thus, the tactic `show x ∈ B from h1 h3.left` will complete the first goal. Once we type this (indented to the same position as the `done` for the first goal), the red squiggle disappears from the first `done`, and the tactic state shows the `No goals` message. If we then click on the blank line for the second goal, the Infoview pane shows the tactic state for that goal:

3.4. Proofs Involving Conjunctions and Biconditionals

Lean File

```
theorem Like_Example_3_4_1 (U : Type)
  (A B C D : Set U) (h1 : A ⊆ B)
  (h2 : ¬∃ (c : U), c ∈ C ∩ D) :
  A ∩ C ⊆ B \ D := by
  define
  fix x : U
  assume h3 : x ∈ A ∩ C
  define at h3; define
  apply And.intro
  · -- Proof that x ∈ B:
    show x ∈ B from h1 h3.left
    done
  · -- Proof that x ∉ D

  done
done
```

Tactic State in Infoview

```
▼ case right
U : Type
A B C D : Set U
h1 : A ⊆ B
h2 : ¬∃ (c : U), c ∈ C ∩ D
x : U
h3 : x ∈ A ∧ x ∈ C
↑ x ∉ D
```

The second goal is a negative statement, and the given `h2` is also a negative statement. This suggests using proof by contradiction, and achieving the contradiction by contradicting `h2`.

Lean File

```
theorem Like_Example_3_4_1 (U : Type)
  (A B C D : Set U) (h1 : A ⊆ B)
  (h2 : ¬∃ (c : U), c ∈ C ∩ D) :
  A ∩ C ⊆ B \ D := by
  define
  fix x : U
  assume h3 : x ∈ A ∩ C
  define at h3; define
  apply And.intro
  · -- Proof that x ∈ B.
    show x ∈ B from h1 h3.left
    done
  · -- Proof that x ∉ D.
    contradict h2 with h4
    done
done
```

Tactic State in Infoview

```
▼ case right
U : Type
A B C D : Set U
h1 : A ⊆ B
h2 : ¬∃ (c : U), c ∈ C ∩ D
x : U
h3 : x ∈ A ∧ x ∈ C
h4 : x ∈ D
↑ ∃ (c : U), c ∈ C ∩ D
```

The goal is now an existential statement, and looking at `h3` and `h4` it is clear that the right value to plug in for `c` in the goal is `x`. The tactic `apply Exists.intro x` will change the goal to `x ∈ C ∩ D` (we have again left off the unnecessary blank at the end of the `apply` tactic).

3.4. Proofs Involving Conjunctions and Biconditionals

Lean File

```
theorem Like_Example_3_4_1 (U : Type)
  (A B C D : Set U) (h1 : A ⊆ B)
  (h2 : ¬∃ (c : U), c ∈ C ∩ D) :
  A ∩ C ⊆ B \ D := by
  define
  fix x : U
  assume h3 : x ∈ A ∩ C
  define at h3; define
  apply And.intro
  · -- Proof that x ∈ B.
    show x ∈ B from h1 h3.left
    done
  · -- Proof that x ∉ D.
    contradict h2 with h4
    apply Exists.intro x
    done
done
```

Tactic State in Infoview

```
▼ case right
U : Type
A B C D : Set U
h1 : A ⊆ B
h2 : ¬∃ (c : U), c ∈ C ∩ D
x : U
h3 : x ∈ A ∧ x ∈ C
h4 : x ∈ D
↑ x ∈ C ∩ D
```

The `define` tactic would now rewrite the goal as $x \in C \wedge x \in D$, and we could prove this goal by combining `h3.right` and `h4`, using the `And.intro` rule. But since we know what the result of the `define` tactic will be, there is really no need to use it. We can just use `And.intro` right away to complete the proof.

Lean File

```
theorem Like_Example_3_4_1 (U : Type)
  (A B C D : Set U) (h1 : A ⊆ B)
  (h2 : ¬∃ (c : U), c ∈ C ∩ D) :
  A ∩ C ⊆ B \ D := by
  define
  fix x : U
  assume h3 : x ∈ A ∩ C
  define at h3; define
  apply And.intro
  · -- Proof that x ∈ B.
    show x ∈ B from h1 h3.left
    done
  · -- Proof that x ∉ D.
    contradict h2 with h4
    apply Exists.intro x
    show x ∈ C ∩ D from And.intro h3.right h4
    done
done
```

Tactic State in Infoview

No goals

Since $P \leftrightarrow Q$ is shorthand for $(P \rightarrow Q) \wedge (Q \rightarrow P)$, the strategies given above for conjunctions lead immediately to the following strategies for biconditionals (*HTPI* p. 132):

To prove a goal of the form $P \leftrightarrow Q$:

Prove $P \rightarrow Q$ and $Q \rightarrow P$ separately.

To use a given of the form $P \leftrightarrow Q$:

Treat this as two separate givens: $P \rightarrow Q$, and $Q \rightarrow P$.

The methods for using these strategies in Lean are similar to those we used above for conjunctions. If we have $h1 : P \rightarrow Q$ and $h2 : Q \rightarrow P$, then `Iff.intro h1 h2` is a proof of $P \leftrightarrow Q$. Thus, if the goal is $P \leftrightarrow Q$, then the tactic `apply Iff.intro _ _` will convert this into two separate goals, $P \rightarrow Q$ and $Q \rightarrow P$. Once again, you can fill in one of these blanks if you already have a proof of either $P \rightarrow Q$ or $Q \rightarrow P$, and you can leave out any blanks at the end of the tactic. If you have a given $h : P \leftrightarrow Q$, then `h.ltr` is a proof of the left-to-right direction of the biconditional, $P \rightarrow Q$, and `h.rtl` is a proof of the right-to-left direction, $Q \rightarrow P$.

Let's try these strategies out in an example.

3.4. Proofs Involving Conjunctions and Biconditionals

Lean File

```
example (U : Type) (P Q : Pred U)
  (h1 : ∀ (x : U), P x ↔ Q x) :
  (exists (x : U), P x) ↔ exists (x : U), Q x := by
    done
```

Tactic State in Infoview

```
U : Type
P Q : Pred U
h1 : ∀ (x : U), P x ↔ Q x
↑ (exists (x : U), P x) ↔
  exists (x : U), Q x
```

The goal is a biconditional statement, so we begin with the tactic `apply Iff.intro`.

Lean File

```
example (U : Type) (P Q : Pred U)
  (h1 : ∀ (x : U), P x ↔ Q x) :
  (exists (x : U), P x) ↔ exists (x : U), Q x := by
  apply Iff.intro
  done
```

Tactic State in Infoview

```
▼ case mp
U : Type
P Q : Pred U
h1 : ∀ (x : U), P x ↔ Q x
↑ (exists (x : U), P x) →
  exists (x : U), Q x
▼ case mpr
U : Type
P Q : Pred U
h1 : ∀ (x : U), P x ↔ Q x
↑ (exists (x : U), Q x) →
  exists (x : U), P x
```

Once again, we have two goals. (The case labels this time aren't very intuitive; "mp" stands for "modus ponens" and "mpr" stands for "modus ponens reverse".) Whenever we have multiple goals, we'll use the bulleted-and-indented style introduced in the last example. As in *HTPI*, we'll label the proofs of the two goals with (\rightarrow) and (\leftarrow) , representing the two directions of the biconditional symbol \leftrightarrow . (You can type \leftarrow in VS Code by typing \l, short for "left".) The first goal is a conditional statement, so we assume the antecedent.

Lean File

```
example (U : Type) (P Q : Pred U)
  (h1 : ∀ (x : U), P x ↔ Q x) :
  (exists (x : U), P x) ↔ exists (x : U), Q x := by
  apply Iff.intro
  · -- (→)
    assume h2 : exists (x : U), P x
    done
  · -- (←)
    done
done
```

Tactic State in Infoview

```
▼ case mp
U : Type
P Q : Pred U
h1 : ∀ (x : U), P x ↔ Q x
h2 : exists (x : U), P x
↑ exists (x : U), Q x
```

3.4. Proofs Involving Conjunctions and Biconditionals

As usual, when we have an existential given, we use it right away.

Lean File

```
example (U : Type) (P Q : Pred U)
  (h1 : ∀ (x : U), P x ↔ Q x) :
  (∃ (x : U), P x) ↔ ∃ (x : U), Q x := by
  apply Iff.intro
  · -- (→)
    assume h2 : ∃ (x : U), P x
    obtain (u : U) (h3 : P u) from h2
    done
  · -- (←)

  done
done
```

Tactic State in Infoview

```
▼ case mp
U : Type
P Q : Pred U
h1 : ∀ (x : U), P x ↔ Q x
h2 : ∃ (x : U), P x
u : U
h3 : P u
↑ ∃ (x : U), Q x
```

Now that we have an object of type U in the tactic state, we can use $h1$ by applying universal instantiation.

Lean File

```
example (U : Type) (P Q : Pred U)
  (h1 : ∀ (x : U), P x ↔ Q x) :
  (∃ (x : U), P x) ↔ ∃ (x : U), Q x := by
  apply Iff.intro
  · -- (→)
    assume h2 : ∃ (x : U), P x
    obtain (u : U) (h3 : P u) from h2
    have h4 : P u ↔ Q u := h1 u
    done
  · -- (←)

  done
done
```

Tactic State in Infoview

```
▼ case mp
U : Type
P Q : Pred U
h1 : ∀ (x : U), P x ↔ Q x
h2 : ∃ (x : U), P x
u : U
h3 : P u
h4 : P u ↔ Q u
↑ ∃ (x : U), Q x
```

Looking at $h3$ and $h4$, we can now see that we will be able to complete the proof if we assign the value u to x in the goal. So our next step is the tactic `apply Exists.intro u`.

3.4. Proofs Involving Conjunctions and Biconditionals

Lean File

```
example (U : Type) (P Q : Pred U)
  (h1 : ∀ (x : U), P x ↔ Q x) :
  (exists (x : U), P x) ↔ exists (x : U), Q x := by
  apply Iff.intro
  · -- (→)
    assume h2 : exists (x : U), P x
    obtain (u : U) (h3 : P u) from h2
    have h4 : P u ↔ Q u := h1 u
    apply Exists.intro u
    done
  · -- (←)
    done
done
```

Tactic State in Infoview

▼ **case mp**
U : Type
P Q : Pred U
h1 : ∀ (x : U), P x ↔ Q x
h2 : exists (x : U), P x
u : U
h3 : P u
h4 : P u ↔ Q u
 $\vdash Q u$

To complete the proof, we use the left-to-right direction of **h4**. We have **h4.ltr** : $P u \rightarrow Q u$ and **h3** : $P u$, so by modus ponens, **h4.ltr** **h3** proves the goal $Q u$. Once we enter this step, Lean indicates that the left-to-right proof is complete, and we can position the cursor below the right-to-left bullet to see the tactic state for the second half of the proof.

Lean File

```
example (U : Type) (P Q : Pred U)
  (h1 : ∀ (x : U), P x ↔ Q x) :
  (exists (x : U), P x) ↔ exists (x : U), Q x := by
  apply Iff.intro
  · -- (→)
    assume h2 : exists (x : U), P x
    obtain (u : U) (h3 : P u) from h2
    have h4 : P u ↔ Q u := h1 u
    apply Exists.intro u
    show Q u from h4.ltr h3
    done
  · -- (←)
    done
done
```

Tactic State in Infoview

▼ **case mpr**
U : Type
P Q : Pred U
h1 : ∀ (x : U), P x ↔ Q x
 $\vdash (\exists (x : U), Q x) \rightarrow$
 $\exists (x : U), P x$

The second half of the proof is similar to the first. We begin by assuming $h2 : \exists (x : U), Q x$, and then we use that assumption to obtain $u : U$ and $h3 : Q u$.

3.4. Proofs Involving Conjunctions and Biconditionals

Lean File

```
example (U : Type) (P Q : Pred U)
  (h1 : ∀ (x : U), P x ↔ Q x) :
  (Ǝ (x : U), P x) ↔ Ǝ (x : U), Q x := by
  apply Iff.intro
  · -- (→)
    assume h2 : Ǝ (x : U), P x
    obtain (u : U) (h3 : P u) from h2
    have h4 : P u ↔ Q u := h1 u
    apply Exists.intro u
    show Q u from h4.ltr h3
    done
  · -- (←)
    assume h2 : Ǝ (x : U), Q x
    obtain (u : U) (h3 : Q u) from h2
    done
done
```

Tactic State in Infoview

```
▼ case mpr
U : Type
P Q : Pred U
h1 : ∀ (x : U), P x ↔ Q x
h2 : Ǝ (x : U), Q x
u : U
h3 : Q u
↑ Ǝ (x : U), P x
```

We can actually shorten the proof by packing a lot into a single step. See if you can figure out the last line of the completed proof below; we'll give an explanation after the proof.

```
example (U : Type) (P Q : Pred U)
  (h1 : ∀ (x : U), P x ↔ Q x) :
  (Ǝ (x : U), P x) ↔ Ǝ (x : U), Q x := by
  apply Iff.intro
  · -- (→)
    assume h2 : Ǝ (x : U), P x
    obtain (u : U) (h3 : P u) from h2
    have h4 : P u ↔ Q u := h1 u
    apply Exists.intro u
    show Q u from h4.ltr h3
    done
  · -- (←)
    assume h2 : Ǝ (x : U), Q x
    obtain (u : U) (h3 : Q u) from h2
    show Ǝ (x : U), P x from Exists.intro u ((h1 u).rtl h3)
    done
done
```

To understand the last step, start with the fact that $h1\ u$ is a proof of $P\ u \leftrightarrow Q\ u$. Therefore $(h1\ u).\text{rtl}$ is a proof of $Q\ u \rightarrow P\ u$, so by modus ponens, $(h1\ u).\text{rtl}\ h3$ is a proof of $P\ u$. It follows that $\text{Exists.intro}\ u\ ((h1\ u).\text{rtl}\ h3)$ is a proof of $\exists\ (x : U), P\ x$, which was the goal.

3.4. Proofs Involving Conjunctions and Biconditionals

There is one more style of reasoning that is sometimes used in proofs of biconditional statements. It is illustrated in Example 3.4.5 of *HTPI*. Here is that theorem, as it is presented in *HTPI* (*HTPI* p. 137).

Theorem. Suppose A , B , and C are sets. Then $A \cap (B \setminus C) = (A \cap B) \setminus C$.

Proof. Let x be arbitrary. Then

$$\begin{aligned} x \in A \cap (B \setminus C) &\text{ iff } x \in A \wedge x \in B \setminus C \\ &\text{ iff } x \in A \wedge x \in B \wedge x \notin C \\ &\text{ iff } x \in (A \cap B) \wedge x \notin C \\ &\text{ iff } x \in (A \cap B) \setminus C. \end{aligned}$$

Thus, $\forall x(x \in A \cap (B \setminus C) \leftrightarrow x \in (A \cap B) \setminus C)$, so $A \cap (B \setminus C) = (A \cap B) \setminus C$. \square

This proof is based on a fundamental principle of set theory that says that if two sets have exactly the same elements, then they are equal. This principle is called the *axiom of extensionality*, and it is what justifies the inference, in the last sentence, from $\forall x(x \in A \cap (B \setminus C) \leftrightarrow x \in (A \cap B) \setminus C)$ to $A \cap (B \setminus C) = (A \cap B) \setminus C$.

The heart of the proof is a string of equivalences that, taken together, establish the biconditional statement $x \in A \cap (B \setminus C) \leftrightarrow x \in (A \cap B) \setminus C$. One can also use this technique to prove a biconditional statement in Lean. This time we'll simply present the complete proof first, and then explain it afterwards.

```
theorem Example_3_4_5 (U : Type)
  (A B C : Set U) : A ∩ (B \ C) = (A ∩ B) \ C := by
  apply Set.ext
  fix x : U
  show x ∈ A ∩ (B \ C) ↔ x ∈ (A ∩ B) \ C from
  calc x ∈ A ∩ (B \ C)
    _ ↔ x ∈ A ∧ (x ∈ B ∧ x ∉ C) := Iff.refl _
    _ ↔ (x ∈ A ∧ x ∈ B) ∧ x ∉ C := and_assoc.symm
    _ ↔ x ∈ (A ∩ B) \ C := Iff.refl _
```

done

The name of the axiom of extensionality in Lean is `Set.ext`, and it is applied in the first step of the Lean proof. As usual, the `apply` tactic works backwards from the goal. In other words, after the first line of the proof, the goal is $\forall (x : U), x \in A \cap (B \setminus C) \leftrightarrow x \in (A \cap B) \setminus C$, because by `Set.ext`, the conclusion of the theorem would follow from this statement. The rest of the proof then proves this goal by introducing an arbitrary x of type U and then proving the biconditional by stringing together several equivalences, exactly as in the *HTPI* proof.

3.4. Proofs Involving Conjunctions and Biconditionals

The proof of the biconditional is called a *calculational proof*, and it is introduced by the keyword `calc`. The calculational proof consists of a string of biconditional statements, each of which is provided with a proof. You can think of the underscore on the left side of each biconditional as standing for the right side of the previous biconditional (or, in the case of the first biconditional, the statement after `calc`).

The proofs of the individual biconditionals in the calculational proof require some explanation. Lean has a large library of theorems that it knows, and you can use those theorems in your proofs. In particular, `Iff.refl` and `and_assoc` are names of theorems in Lean's library. You can find out what any theorem says by using the Lean command `#check`. (Commands that ask Lean for a response generally start with the character `#`.) If you type `#check Iff.refl` in a Lean file, you will see Lean's response in the Infoview pane: `Iff.refl (a : Prop) : a ↔ a`. What this tells us is that Lean already knows the theorem

```
theorem Iff.refl (a : Prop) : a ↔ a
```

(This theorem says that “iff” has a property called *reflexivity*; we'll discuss reflexivity in Chapter 4.) When variables are declared in the statement of a theorem, it is understood that they can stand for anything of the appropriate type (see Section 3.1 of *HPTI*). Thus, the theorem `Iff.refl` can be thought of as establishing the truth of the statement $\forall (a : \text{Prop}), a \leftrightarrow a$. In fact, you can get Lean to report the meaning of the theorem in this form with the command `#check @Iff.refl`. What this means is that, in any proof, Lean lets you treat `Iff.refl` as a proof of the statement $\forall (a : \text{Prop}), a \leftrightarrow a$. Thus, by universal instantiation, for any proposition `a`, Lean will recognize `Iff.refl a` as a proof of `a ↔ a`. This is used to justify the first biconditional in the calculational proof.

But wait! The first biconditional in the calculational proof is $x \in A \cap (B \setminus C) \leftrightarrow x \in A \wedge (x \in B \wedge x \notin C)$, which does not have the form `a ↔ a`. How can it be justified by the theorem `Iff.refl`? Recall that Lean doesn't need to be told to write out definitions of mathematical notation; it does that automatically. When the definitions of the set theory notation are written out, the first biconditional in the calculational proof becomes $x \in A \wedge (x \in B \wedge x \notin C) \leftrightarrow x \in A \wedge (x \in B \wedge x \notin C)$, which *does* have the form `a ↔ a`, so it can be proven with the term-mode proof `Iff.refl _`. Note that we are using an underscore here to ask Lean to figure out what to plug in for `a`. This saves us the trouble of writing out the full term-mode proof, which would be `Iff.refl (x ∈ A ∧ (x ∈ B ∧ x ∉ C))`. The lesson of this example is that the theorem `Iff.refl` is more powerful than it looks. Not only can we use `Iff.refl _` to prove statements of the form `a ↔ a`, we can also use it to prove statements of the form `a ↔ a'`, if `a` and `a'` reduce to the same thing when definitions are filled in. We say in this case that `a` and `a'` are *definitionally equal*. This explains the third line of the calculational proof, which is also justified by the proof `Iff.refl _`.

The second line uses the theorem `and_assoc`. If you type `#check and_assoc`, you will get this response from Lean:

3.4. Proofs Involving Conjunctions and Biconditionals

```
and_assoc {a b c : Prop} : (a ∧ b) ∧ c ↔ a ∧ b ∧ c
```

Once again, it is understood that the variables a , b , and c can stand for any propositions, as you can see by giving the command `#check @and_assoc`. This generates the response

```
@and_assoc : ∀ {a b c : Prop}, (a ∧ b) ∧ c ↔ a ∧ b ∧ c
```

which is shorthand for

```
@and_assoc : ∀ {a : Prop}, ∀ {b : Prop}, ∀ {c : Prop},
(a ∧ b) ∧ c ↔ a ∧ (b ∧ c)
```

Recall that the curly braces indicate that a , b , and c are implicit arguments, and that Lean groups the logical connectives to the right, which means that it interprets $a \wedge b \wedge c$ as $a \wedge (b \wedge c)$. This is the associative law for “and” (see Section 1.2 of *HTPI*). Since a , b , and c are implicit, Lean will recognize `and_assoc` as a proof of any statement of the form $(a \wedge b) \wedge c \leftrightarrow a \wedge (b \wedge c)$, where a , b , and c can be replaced with any propositions. Lean doesn’t need to be told what propositions are being used as a , b , and c ; it will figure that out for itself. Unfortunately, the second biconditional in the calculational proof is $x \in A \wedge (x \in B \wedge x \notin C) \leftrightarrow (x \in A \wedge x \in B) \wedge x \notin C$, which has the form $a \wedge (b \wedge c) \leftrightarrow (a \wedge b) \wedge c$, not $(a \wedge b) \wedge c \leftrightarrow a \wedge (b \wedge c)$. (Notice that the first of these biconditionals is the same as the second except that the left and right sides have been swapped.) To account for this discrepancy, we use the fact that if h is a proof of any biconditional $P \leftrightarrow Q$, then $h.\text{symm}$ is a proof of $Q \leftrightarrow P$. Thus `and_assoc.symm` proves the second biconditional in the calculational proof. (By the way, the *HTPI* proof avoids any mention of the associativity of “and” by simply leaving out parentheses in the conjunction $x \in A \wedge x \in B \wedge x \notin C$. As explained in Section 1.2 of *HTPI*, this represents an implicit use of the associativity of “and.”)

You can get a better understanding of the first step of our last proof by typing `#check @Set.ext`. The result is

```
@Set.ext : ∀ {α : Type u_1} {a b : Set α},
(∀ (x : α), x ∈ a ↔ x ∈ b) → a = b
```

which is shorthand for

```
@Set.ext : ∀ {α : Type u_1}, ∀ {a : Set α}, ∀ {b : Set α},
(∀ (x : α), x ∈ a ↔ x ∈ b) → a = b
```

Ignoring the u_1 , whose significance won’t be important to us, this means that `Set.ext` can be used to prove any statement of the form $(\forall (x : \alpha), x \in a \leftrightarrow x \in b) \rightarrow a = b$, where α can be replaced by any type and a and b can be replaced by any sets of objects of type α . Make sure you understand how this explains the effect of the tactic `apply Set.ext` in the first step of our last proof. Almost all of our proofs that two sets are equal will start with `apply Set.ext`.

3.4. Proofs Involving Conjunctions and Biconditionals

Notice that in Lean's responses to both `#check @and_assoc` and `#check @Set.ext`, multiple universal quantifiers in a row were grouped together and written as a single universal quantifier followed by a list of variables (with types). Lean allows this notational shorthand for any sequence of consecutive quantifiers, as long as they are all of the same kind (all existential or all universal), and we will use this notation from now on.

Exercises

1. `theorem Exercise_3_4_2 (U : Type) (A B C : Set U)
(h1 : A ⊆ B) (h2 : A ⊆ C) : A ⊆ B ∩ C := by`

done

2. `theorem Exercise_3_4_4 (U : Type) (A B C : Set U)
(h1 : A ⊆ B) (h2 : A ⊈ C) : B ⊈ C := by`

done

3. `theorem Exercise_3_3_12 (U : Type)
(F G : Set (Set U)) : F ⊆ G → U₀ F ⊆ U₀ G := by`

done

4. `theorem Exercise_3_3_16 (U : Type) (B : Set U)
(F : Set (Set U)) : F ⊆ ℙ B → U₀ F ⊆ B := by`

done

5. `theorem Exercise_3_3_17 (U : Type) (F G : Set (Set U))
(h1 : ∀ (A : Set U), A ∈ F → ∀ (B : Set U), B ∈ G → A ⊆ B) :
U₀ F ⊆ U₀ G := by`

done

6. `theorem Exercise_3_4_7 (U : Type) (A B : Set U) :
ℳ (A ∩ B) = ℳ A ∩ ℳ B := by`

done

7. `theorem Exercise_3_4_17 (U : Type) (A : Set U) : A = U₀ ℳ A := by`

done

3.5. Proofs Involving Disjunctions

8. `theorem Exercise_3_4_18a (U : Type) (F G : Set (Set U)) :`
 $\cup_0 (F \cap G) \subseteq (\cup_0 F) \cap (\cup_0 G) := \text{by}$
- done
9. `theorem Exercise_3_4_19 (U : Type) (F G : Set (Set U)) :`
 $(\cup_0 F) \cap (\cup_0 G) \subseteq \cup_0 (F \cap G) \leftrightarrow$
 $\forall (A B : \text{Set } U), A \in F \rightarrow B \in G \rightarrow A \cap B \subseteq \cup_0 (F \cap G) := \text{by}$
- done

3.5. Proofs Involving Disjunctions

A common proof method for dealing with givens or goals that are disjunctions is *proof by cases*. Here's how it works (*HTPI* p. 143).

To use a given of the form $P \vee Q$:

Break your proof into cases. For case 1, assume that P is true and use this assumption to prove the goal. For case 2, assume that Q is true and prove the goal.

In Lean, you can break a proof into cases by using the `by_cases` tactic. If you have a given $h : P \vee Q$, then the tactic `by_cases` on h will break your proof into two cases. For the first case, the given h will be changed to $h : P$, and for the second, it will be changed to $h : Q$; the goal for both cases will be the same as the original goal. Thus, the effect of the `by_cases` on h tactic is as follows:

Tactic State Before Using Strategy

```
:
h : P ∨ Q
⊢ goal
```

Tactic State After Using Strategy

```
▼ case Case_1
:
h : P
⊢ goal
▼ case Case_2
:
h : Q
⊢ goal
```

Notice that the original given $h : P \vee Q$ gets *replaced* by $h : P$ in case 1 and $h : Q$ in case 2. This is usually what is most convenient, but if you write `by_cases` on h with $h1$, then the original given h will be preserved, and new givens $h1 : P$ and $h1 : Q$ will be added to cases 1

3.5. Proofs Involving Disjunctions

and 2, respectively. If you want different names for the new givens in the two cases, then use `by_cases on h with h1, h2` to add the new given `h1 : P` in case 1 and `h2 : Q` in case 2.

You can follow `by_cases on` with any proof of a disjunction, even if that proof is not just a single identifier. In that cases you will want to add `with` to specify the identifier or identifiers to be used for the new assumptions in the two cases. Another variant is that you can use the tactic `by_cases h : P` to break your proof into two cases, with the new assumptions being `h : P` in case 1 and `h : ¬P` in case 2. In other words, the effect of `by_cases h : P` is the same as adding the new given `h : P ∨ ¬P` (which, of course, is a tautology) and then using the tactic `by_cases on h`.

There are several introduction rules that you can use in Lean to prove a goal of the form `P ∨ Q`. If you have `h : P`, then Lean will accept `Or.intro_left Q h` as a proof of `P ∨ Q`. In most situations Lean can infer the proposition `Q` from context, and in that case you can use the shorter form `Or.inl h` as a proof of `P ∨ Q`. You can see the difference between `Or.intro_left` and `Or.inl` by using the `#check` command:

```
@Or.intro_left : ∀ {a : Prop} (b : Prop), a → a ∨ b
```

```
@Or.inl : ∀ {a b : Prop}, a → a ∨ b
```

Notice that `b` is an implicit argument in `Or.inl`, but not in `Or.intro_left`.

Similarly, if you have `h : Q`, then `Or.intro_right P h` is a proof of `P ∨ Q`. In most situations Lean can infer `P` from context, and you can use the shorter form `Or.inr h`.

Often, when your goal has the form `P ∨ Q`, you will be unable to prove `P`, and also unable to prove `Q`. Proof by cases can help in that situation as well (*HTPI* p. 145).

To prove a goal of the form `P ∨ Q`:

Break your proof into cases. In each case, either prove `P` or prove `Q`.

Example 3.5.2 from *HTPI* illustrates these strategies:

Lean File

```
theorem Example_3_5_2
  (U : Type) (A B C : Set U) :
  A \ (B \ C) ⊆ (A \ B) ∪ C := by
```

```
done
```

Tactic State in Infoview

<code>U : Type</code> <code>A B C : Set U</code> <code>↑ A \ (B \ C) ⊆ A \ B ∪ C</code>

The `define` tactic would rewrite the goal as `∀ {a : U}, a ∈ A \ (B \ C) → a ∈ A \ B ∪ C`, which suggests that our next two tactics should be `fix x : U` and `assume h1 : x ∈ A \ (B \ C)`. But as we have seen before, if you know what the result of the `define` tactic is going to be,

3.5. Proofs Involving Disjunctions

then there is usually no need to use it. After introducing x as an arbitrary element of $A \setminus (B \setminus C)$, we write out the definitions of our new given and goal to help guide our next strategy choice:

Lean File

```
theorem Example_3_5_2
  (U : Type) (A B C : Set U) :
  A \ (B \ C) ⊆ (A \ B) ∪ C := by
  fix x : U
  assume h1 : x ∈ A \ (B \ C)
  define; define at h1
  done
```

Tactic State in Infoview

$\text{U} : \text{Type}$ $\text{A B C} : \text{Set U}$ $x : U$ $\text{h1} : x \in A \wedge x \notin B \setminus C$ $\vdash x \in A \setminus B \vee x \in C$
--

The goal is now a disjunction, which suggests that proof by cases might be helpful. But what cases should we use? The key is to look at the meaning of the right half of the given $h1$. The meaning of $x \notin B \setminus C$ is $\neg(x \in B \wedge x \notin C)$, which, by one of the De Morgan laws, is equivalent to $x \notin B \vee x \in C$.

Lean File

```
theorem Example_3_5_2
  (U : Type) (A B C : Set U) :
  A \ (B \ C) ⊆ (A \ B) ∪ C := by
  fix x : U
  assume h1 : x ∈ A \ (B \ C)
  define; define at h1
  have h2 : x ∉ B \ C := h1.right
  define at h2; demorgan at h2
    --h2 : x ∉ B ∨ x ∈ C
  done
```

Tactic State in Infoview

$\text{U} : \text{Type}$ $\text{A B C} : \text{Set U}$ $x : U$ $\text{h1} : x \in A \wedge x \notin B \setminus C$ $\text{h2} : x \notin B \vee x \in C$ $\vdash x \in A \setminus B \vee x \in C$

The new given $h2$ is now a disjunction, which suggests what cases we should use:

Lean File

```
theorem Example_3_5_2
  (U : Type) (A B C : Set U) :
  A \ (B \ C) ⊆ (A \ B) ∪ C := by
  fix x : U
  assume h1 : x ∈ A \ (B \ C)
  define; define at h1
  have h2 : x ∉ B \ C := h1.right
  define at h2; demorgan at h2
    --h2 : x ∉ B ∨ x ∈ C
  by_cases on h2
  done
```

Tactic State in Infoview

```
▼ case Case_1
U : Type
A B C : Set U
x : U
h1 : x ∈ A ∧ x ∉ B \ C
h2 : x ∉ B
↑ x ∈ A \ B ∨ x ∈ C
▼ case Case_2
U : Type
A B C : Set U
x : U
h1 : x ∈ A ∧ x ∉ B \ C
h2 : x ∈ C
↑ x ∈ A \ B ∨ x ∈ C
```

Of course, now that we have two goals, we will introduce bullets labeling the two parts of the proof as case 1 and case 2. Looking at the givens `h1` and `h2` in both cases, it is not hard to see that we should be able to prove $x \in A \setminus B$ in case 1 and $x \in C$ in case 2. Thus, in case 1 we will be able to give a proof of the goal that has the form `Or.inl _`, where the blank will be filled in with a proof of $x \in A \setminus B$, and in case 2 we can use `Or.inr _`, filling in the blank with a proof of $x \in C$. This suggests that we should use the tactics `apply Or.inl` in case 1 and `apply Or.inr` in case 2. Focusing first on case 1, we get:

Lean File

```
theorem Example_3_5_2
  (U : Type) (A B C : Set U) :
  A \ (B \ C) ⊆ (A \ B) ∪ C := by
  fix x : U
  assume h1 : x ∈ A \ (B \ C)
  define; define at h1
  have h2 : x ∉ B \ C := h1.right
  define at h2; demorgan at h2
    --h2 : x ∉ B ∨ x ∈ C
  by_cases on h2
  · -- Case 1. h2 : x ∉ B
    apply Or.inl
    done
  · -- Case 2. h2 : x ∈ C
    done
done
```

Tactic State in Infoview

```
▼ case Case_1.h
U : Type
A B C : Set U
x : U
h1 : x ∈ A ∧ x ∉ B \ C
h2 : x ∉ B
↑ x ∈ A \ B
```

3.5. Proofs Involving Disjunctions

Notice that the tactic `apply Or.inl` has changed the goal for case 1 to the left half of the original goal, $x \in A \setminus B$. Since this means $x \in A \wedge x \notin B$, we can complete case 1 by combining `h1.left` with `h2`, and then we can move on to case 2.

Lean File

```
theorem Example_3_5_2
  (U : Type) (A B C : Set U) :
  A \ (B \ C) ⊆ (A \ B) ∪ C := by
  fix x : U
  assume h1 : x ∈ A \ (B \ C)
  define; define at h1
  have h2 : x ∉ B \ C := h1.right
  define at h2; demorgan at h2
    --h2 : x ∉ B ∨ x ∈ C
  by_cases on h2
  · -- Case 1. h2 : x ∉ B
    apply Or.inl
    show x ∈ A \ B from And.intro h1.left h2
    done
  · -- Case 2. h2 : x ∈ C
    done
done
```

Tactic State in Infoview

```
▼case Case_2
U : Type
A B C : Set U
x : U
h1 : x ∈ A ∧ x ∉ B \ C
h2 : x ∈ C
↑ x ∈ A \ B ∨ x ∈ C
```

Case 2 is similar, using `Or.inr` and `h2`

Lean File

```
theorem Example_3_5_2
  (U : Type) (A B C : Set U) :
  A \ (B \ C) ⊆ (A \ B) ∪ C := by
  fix x : U
  assume h1 : x ∈ A \ (B \ C)
  define; define at h1
  have h2 : x ∉ B \ C := h1.right
  define at h2; demorgan at h2
    --h2 : x ∉ B ∨ x ∈ C
  by_cases on h2
  · -- Case 1. h2 : x ∉ B
    apply Or.inl
    show x ∈ A \ B from And.intro h1.left h2
    done
  · -- Case 2. h2 : x ∈ C
    apply Or.inr
    show x ∈ C from h2
    done
  done
```

Tactic State in Infoview**No goals**

There is a second strategy that is often useful to prove a goal of the form $P \vee Q$. It is motivated by the fact that $P \vee Q$ is equivalent to both $\neg P \rightarrow Q$ and $\neg Q \rightarrow P$ (*HTPI* p. 147).

To prove a goal of the form $P \vee Q$:

Assume that P is false and prove Q , or assume that Q is false and prove P .

If your goal is $P \vee Q$, then the Lean tactic `or_left` with h will add the new given $h : \neg Q$ to the tactic state and set the goal to be P , and `or_right` with h will add $h : \neg P$ to the tactic state and set the goal to be Q . For example, here is the effect of the tactic `or_left` with h :

Tactic State Before Using Strategy

```
:
└ P ∨ Q
```

Tactic State After Using Strategy

```
:
h : ¬Q
└ P
```

Notice that `or_left` and `or_right` have the same effect as `apply Or.inl` and `apply Or.inr`, except that each adds a new given to the tactic state. Sometimes you can tell in advance that you won't need the extra given, and in that case the tactics `apply Or.inl` and `apply Or.inr` can be useful. For example, that was the case in the example above. But if you think the

3.5. Proofs Involving Disjunctions

extra given might be useful, you are better off using `or_left` or `or_right`. Here's an example illustrating this.

Lean File

```
example (U : Type) (A B C : Set U)
      (h1 : A \ B ⊆ C) : A ⊆ B ∪ C := by
      done
```

Tactic State in Infoview

U : Type
A B C : Set U
h1 : A \ B ⊆ C
$\vdash A \subseteq B \cup C$

Of course, we begin by letting x be an arbitrary element of A . Writing out the meaning of the new goal shows that it is a disjunction.

Lean File

```
example (U : Type) (A B C : Set U)
      (h1 : A \ B ⊆ C) : A ⊆ B ∪ C := by
      fix x : U
      assume h2 : x ∈ A
      define
      done
```

Tactic State in Infoview

U : Type
A B C : Set U
h1 : A \ B ⊆ C
x : U
h2 : x ∈ A
$\vdash x \in B \vee x \in C$

Looking at the givens $h1$ and $h2$, we see that if we assume $x \notin B$, then we should be able to prove $x \in C$. This suggests that we should use the `or_right` tactic.

Lean File

```
example (U : Type) (A B C : Set U)
      (h1 : A \ B ⊆ C) : A ⊆ B ∪ C := by
      fix x : U
      assume h2 : x ∈ A
      define
      or_right with h3
      done
```

Tactic State in Infoview

U : Type
A B C : Set U
h1 : A \ B ⊆ C
x : U
h2 : x ∈ A
h3 : x \notin B
$\vdash x \in C$

We can now complete the proof. Notice that $h1 _$ will be a proof of the goal $x \in C$, if we can fill in the blank with a proof of $x \in A \setminus B$. Since $x \in A \setminus B$ means $x \in A \wedge x \notin B$, we can prove it with the expression `And.intro h2 h3`.

Lean File

```
example (U : Type) (A B C : Set U)
  (h1 : A \ B ⊆ C) : A ⊆ B ∪ C := by
  fix x : U
  assume h2 : x ∈ A
  define
  or_right with h3
  show x ∈ C from h1 (And.intro h2 h3)
  done
```

Tactic State in Infoview

No goals

The fact that $P \vee Q$ is equivalent to both $\neg P \rightarrow Q$ and $\neg Q \rightarrow P$ also suggests another strategy for using a given that is a disjunction (*HTPI* p. 149).

To use a given of the form $P \vee Q$:

If you are also given $\neg P$, or you can prove that P is false, then you can use this given to conclude that Q is true. Similarly, if you are given $\neg Q$ or can prove that Q is false, then you can conclude that P is true.

This strategy is a rule of inference called *disjunctive syllogism*, and the tactic for using this strategy in Lean is called `disj_syll`. If you have $h1 : P \vee Q$ and $h2 : \neg P$, then the tactic `disj_syll h1 h2` will change $h1$ to $h1 : Q$; if instead you have $h2 : \neg Q$, then `disj_syll h1 h2` will change $h1$ to $h1 : P$. Notice that, as with the `by_cases` tactic, the given $h1$ gets *replaced* with the conclusion of the rule. The tactic `disj_syll h1 h2 with h3` will preserve the original $h1$ and introduce the conclusion as a new given with the identifier $h3$. Also, as with the `by_cases` tactic, either $h1$ or $h2$ can be a complex proof rather than simply an identifier (although in that case it must be enclosed in parentheses, so that Lean can tell where $h1$ ends and $h2$ begins). The only requirement is that $h1$ must be a proof of a disjunction, and $h2$ must be a proof of the negation of one side of the disjunction. If $h1$ is not simply an identifier, then you will want to use `with` to specify the identifier to be used for the conclusion of the rule.

Here's an example illustrating the use of the disjunctive syllogism rule.

Lean File

```
example (U : Type) (A B C : Set U) (h1 : A ⊆ B ∪ C)
  (h2 : ¬∃ (x : U), x ∈ A ∩ B) : A ⊆ C := by
  done
```

Tactic State in Infoview

$U : \text{Type}$
 $A B C : \text{Set } U$
 $h1 : A \subseteq B \cup C$
 $h2 : \neg \exists (x : U), x \in A \cap B$
 $\vdash A \subseteq C$

Of course, we begin by introducing an arbitrary element of A . We also rewrite $h2$ as an equivalent positive statement.

Lean File

```
example
  (U : Type) (A B C : Set U) (h1 : A ⊆ B ∪ C)
  (h2 : ¬∃ (x : U), x ∈ A ∩ B) : A ⊆ C := by
  fix a : U
  assume h3 : a ∈ A
  quant_neg at h2
  done
```

Tactic State in Infoview

```
U : Type
A B C : Set U
h1 : A ⊆ B ∪ C
h2 : ∀ (x : U),
  x ∉ A ∩ B
a : U
h3 : a ∈ A
  ⊢ a ∈ C
```

We can now make two inferences by combining $h1$ with $h3$ and by applying $h2$ to a . To see how to use the inferred statements, we write out their definitions, and since one of them is a negative statement, we reexpress it as an equivalent positive statement.

Lean File

```
example
  (U : Type) (A B C : Set U) (h1 : A ⊆ B ∪ C)
  (h2 : ¬∃ (x : U), x ∈ A ∩ B) : A ⊆ C := by
  fix a : U
  assume h3 : a ∈ A
  quant_neg at h2
  have h4 : a ∈ B ∪ C := h1 h3
  have h5 : a ∉ A ∩ B := h2 a
  define at h4
  define at h5; demorgan at h5
  done
```

Tactic State in Infoview

```
U : Type
A B C : Set U
h1 : A ⊆ B ∪ C
h2 : ∀ (x : U),
  x ∉ A ∩ B
a : U
h3 : a ∈ A
h4 : a ∈ B ∨ a ∈ C
h5 : a ∉ A ∨ a ∉ B
  ⊢ a ∈ C
```

Both $h4$ and $h5$ are disjunctions, and looking at $h3$ we see that the disjunctive syllogism rule can be applied. From $h3$ and $h5$ we can draw the conclusion $a \notin B$, and then combining that conclusion with $h4$ we can infer $a \in C$. Since that is the goal, we are done.

Lean File

```
example
  (U : Type) (A B C : Set U) (h1 : A ⊆ B ∪ C)
  (h2 : ¬∃ (x : U), x ∈ A ∩ B) : A ⊆ C := by
  fix a : U
  assume h3 : a ∈ A
  quant_neg at h2
  have h4 : a ∈ B ∪ C := h1 h3
  have h5 : a ∉ A ∩ B := h2 a
  define at h4
  define at h5; demorgan at h5
  disj_syll h5 h3 --h5 : a ∉ B
  disj_syll h4 h5 --h4 : a ∈ C
  show a ∈ C from h4
  done
```

Tactic State in Infoview**No goals**

We're going to redo the last example, to illustrate another useful technique in Lean. We start with some of the same steps as before.

Lean File

```
example
  (U : Type) (A B C : Set U) (h1 : A ⊆ B ∪ C)
  (h2 : ¬∃ (x : U), x ∈ A ∩ B) : A ⊆ C := by
  fix a : U
  assume h3 : a ∈ A
  have h4 : a ∈ B ∪ C := h1 h3
  define at h4
  done
```

Tactic State in Infoview

U : Type
A B C : Set U
h1 : A ⊆ B ∪ C
h2 : ¬∃ (x : U),
 x ∈ A ∩ B
a : U
h3 : a ∈ A
h4 : a ∈ B ∨ a ∈ C
 ⊢ a ∈ C

At this point, you might see a possible route to the goal: from **h2** and **h3** we should be able to prove that **a ∉ B**, and then, combining that with **h4** by the disjunctive syllogism rule, we should be able to deduce the goal **a ∈ C**. Let's try writing the proof that way.

Lean File

```
example
  (U : Type) (A B C : Set U) (h1 : A ⊆ B ∪ C)
  (h2 : ¬∃ (x : U), x ∈ A ∩ B) : A ⊆ C := by
  fix a : U
  assume h3 : a ∈ A
  have h4 : a ∈ B ∪ C := h1 h3
  define at h4
  have h5 : a ∉ B := sorry
  disj_syll h4 h5 --h4 : a ∈ C
  show a ∈ C from h4
  done
```

Tactic State in Infoview**No goals**

We have introduced a new idea in this proof. The justification we have given for introducing `h5 : a ∉ B` is `sorry`. You might think of this as meaning “Sorry, I’m not going to give a justification for this statement, but please accept it anyway.” Of course, this is cheating; in a complete proof, every step must be justified. Lean accepts `sorry` as a proof of any statement, but it displays it in red to warn you that you’re cheating. It also puts a brown squiggle under the keyword `example` and a warning symbol in the left margin, and it puts the message `declaration uses 'sorry'` in the Infoview, to warn you that, although the proof has reached the goal, it is not fully justified.

Although writing the proof this way is cheating, it is a convenient way to see that our plan of attack for this proof is reasonable. Lean has accepted the proof, except for the warning that we have used `sorry`. So now we know that if we go back and replace `sorry` with a proof of `a ∉ B`, then we will have a complete proof.

The proof of `a ∉ B` is hard enough that it is easier to do it in tactic mode rather than term mode. So we will begin the proof as we always do for tactic-mode proofs: we replace `sorry` with `by`, leave a blank line, and then put `done`, indented further than the surrounding text. When we put the cursor on the blank line before `done`, we see the tactic state for our “proof within a proof.”

Lean File

```

example
  (U : Type) (A B C : Set U) (h1 : A ⊆ B ∪ C)
  (h2 : ¬∃ (x : U), x ∈ A ∩ B) : A ⊆ C := by
fix a : U
assume h3 : a ∈ A
have h4 : a ∈ B ∪ C := h1 h3
define at h4
have h5 : a ∉ B := by
  done
disj_syll h4 h5 --h4 : a ∈ C
show a ∈ C from h4
done
    
```

Tactic State in Infoview

```

U : Type
A B C : Set U
h1 : A ⊆ B ∪ C
h2 : ¬∃ (x : U),
  x ∈ A ∩ B
a : U
h3 : a ∈ A
h4 : a ∈ B ∨ a ∈ C
  ⊢ a ∉ B
    
```

Note that $h5 : a \notin B$ is not a given in the tactic state, because we have not yet justified it; in fact, $a \notin B$ is the goal. This goal is a negative statement, and $h2$ is also negative. This suggests that we could try using proof by contradiction, achieving the contradiction by contradicting $h2$. So we use the tactic `contradict h2 with h6`.

Lean File

```

example
  (U : Type) (A B C : Set U) (h1 : A ⊆ B ∪ C)
  (h2 : ¬∃ (x : U), x ∈ A ∩ B) : A ⊆ C := by
fix a : U
assume h3 : a ∈ A
have h4 : a ∈ B ∪ C := h1 h3
define at h4
have h5 : a ∉ B := by
  contradict h2 with h6
  done
disj_syll h4 h5 --h4 : a ∈ C
show a ∈ C from h4
done
    
```

Tactic State in Infoview

```

U : Type
A B C : Set U
h1 : A ⊆ B ∪ C
h2 : ¬∃ (x : U),
  x ∈ A ∩ B
a : U
h3 : a ∈ A
h4 : a ∈ B ∨ a ∈ C
h6 : a ∈ B
  ⊢ ∃ (x : U), x ∈ A ∩ B
    
```

Looking at $h3$ and $h6$, we see that the right value to plug in for x in the goal is a . In fact, `Exists.intro a _` will prove the goal, if we can fill in the blank with a proof of $a \in A \cap B$. Since this means $a \in A \wedge a \in B$, we can prove it with `And.intro h3 h6`. Thus, we can complete the proof in one more step:

Lean File

```
example
  (U : Type) (A B C : Set U) (h1 : A ⊆ B ∪ C)
  (h2 : ¬∃ (x : U), x ∈ A ∩ B) : A ⊆ C := by
  fix a : U
  assume h3 : a ∈ A
  have h4 : a ∈ B ∪ C := h1 h3
  define at h4
  have h5 : a ∉ B := by
    contradict h2 with h6
    show ∃ (x : U), x ∈ A ∩ B from
      Exists.intro a (And.intro h3 h6)
    done
  disj_syll h4 h5 --h4 : a ∈ C
  show a ∈ C from h4
  done
```

Tactic State in Infoview**No goals**

The error has disappeared from the word `done` and a check mark has appeared in the left margin next to `example`, indicating that the proof is complete.

It was not really necessary for us to use `sorry` when writing this proof. We could have simply written the steps in order, exactly as they appear above. Any time you use the `have` tactic with a conclusion that is difficult to justify, you have a choice. You can establish the `have` with `sorry`, complete the proof, and then return and fill in a justification for the `have`, as we did in the example above. Or, you can justify the `have` right away by typing `by` after `:=` and then plunging into the “proof within in a proof.” Once you complete the inner proof, you can continue with the original proof.

And in case you were wondering: yes, if the inner proof uses the `have` tactic with a statement that is hard to justify, then you can write a “proof within a proof within a proof”!

Exercises

In each case, replace `sorry` with a proof.

1. `theorem Exercise_3_5_2 (U : Type) (A B C : Set U) :
 (A ∪ B) \ C ⊆ A ∪ (B \ C) := sorry`
2. `theorem Exercise_3_5_5 (U : Type) (A B C : Set U)
 (h1 : A ∩ C ⊆ B ∩ C) (h2 : A ∪ C ⊆ B ∪ C) : A ⊆ B := sorry`
3. `theorem Exercise_3_5_7 (U : Type) (A B C : Set U) :
 A ∪ C ⊆ B ∪ C ↔ A \ C ⊆ B \ C := sorry`

4. `theorem Exercise_3_5_8 (U : Type) (A B : Set U) :`
 $\mathcal{P} A \cup \mathcal{P} B \subseteq \mathcal{P} (A \cup B)$:= `sorry`
5. `theorem Exercise_3_5_17b (U : Type) (F : Set (Set U)) (B : Set U) :`
 $B \cup (\cap_{\emptyset} F) = \{x : U \mid \forall (A : Set U), A \in F \rightarrow x \in B \cup A\}$:= `sorry`
6. `theorem Exercise_3_5_18 (U : Type) (F G H : Set (Set U))`
 $(h1 : \forall (A : Set U), A \in F \rightarrow \forall (B : Set U), B \in G \rightarrow A \cup B \in H) :$
 $\cap_{\emptyset} H \subseteq (\cap_{\emptyset} F) \cup (\cap_{\emptyset} G)$:= `sorry`
7. `theorem Exercise_3_5_24a (U : Type) (A B C : Set U) :`
 $(A \cup B) \Delta C \subseteq (A \Delta C) \cup (B \Delta C)$:= `sorry`

3.6. Existence and Uniqueness Proofs

Recall that $\exists! (x : U), P x$ means that there is exactly one x of type U such that $P x$ is true. One way to deal with a given or goal of this form is to use the `define` tactic to rewrite it as the equivalent statement $\exists (x : U), P x \wedge \forall (x_1 : U), P x_1 \rightarrow x_1 = x$. You can then apply techniques discussed previously in this chapter. However, there are also proof techniques, and corresponding Lean tactics, for working directly with givens and goals of this form.

Often a goal of the form $\exists! (x : U), P x$ is proven by using the following strategy. This is a slight rephrasing of the strategy presented in *HTPI*. The rephrasing is based on the fact that for any propositions A , B , and C , $A \wedge B \rightarrow C$ is equivalent to $A \rightarrow B \rightarrow C$ (you can check this equivalence by making a truth table). The second of these statements is usually easier to work with in Lean than the first one, so we will often rephrase statements that have the form $A \wedge B \rightarrow C$ as $A \rightarrow B \rightarrow C$. To see why the second statement is easier to use, suppose that you have givens $hA : A$ and $hB : B$. If you also have $h : A \rightarrow B \rightarrow C$, then $h hA$ is a proof of $B \rightarrow C$, and therefore $h hA hB$ is a proof of C . If instead you had $h' : (A \wedge B) \rightarrow C$, then to prove C you would have to write $h' (\text{And.intro } hA hB)$, which is a bit less convenient.

With that preparation, here is our strategy for proving statements of the form $\exists! (x : U), P x$ (*HTPI* pp. 156–157).

To prove a goal of the form $\exists! (x : U), P x$:

Prove $\exists (x : U), P x$ and $\forall (x_1 x_2 : U), P x_1 \rightarrow P x_2 \rightarrow x_1 = x_2$. The first of these goals says that there exists an x such that $P x$ is true, and the second says that it is unique. The two parts of the proof are therefore sometimes labeled *existence* and *uniqueness*.

3.6. Existence and Uniqueness Proofs

To apply this strategy in a Lean proof, we use the tactic `exists_unique`. We'll illustrate this with the theorem from Example 3.6.2 in *HTPI*. Here's how that theorem and its proof are presented in *HTPI* (*HTPI* pp. 157–158):

Theorem. *There is a unique set A such that for every set B, $A \cup B = B$.*

Proof. Existence: Clearly $\forall B(\emptyset \cup B = B)$, so \emptyset has the required property.

Uniqueness: Suppose $\forall B(C \cup B = B)$ and $\forall B(D \cup B = B)$. Applying the first of these assumptions to D we see that $C \cup D = D$, and applying the second to C we get $D \cup C = C$. But clearly $C \cup D = D \cup C$, so $C = D$. \square

You will notice that there are two statements in this proof that are described as “clearly” true. This brings up one of the difficulties with proving theorems in Lean: things that are clear to *us* are not necessarily clear to *Lean!* There are two ways to deal with such “clear” statements. The first is to see if the statement is in the library of theorems that Lean knows. The second is to prove the statement as a preliminary theorem that can then be used in the proof of our main theorem. We'll take the second approach here, since proving these “clear” facts will give us more practice with Lean proofs, but later we'll have more to say about searching for statements in Lean's theorem library.

The first theorem we need says that for every set B , $\emptyset \cup B = B$, and it brings up a subtle issue: in Lean, the symbol \emptyset is ambiguous! The reason for this is Lean's strict typing rules. For each type U , there is an empty set of type `Set U`. There is, for example, the set of type `Set Nat` that contains no natural numbers, and also the set of type `Set Real` that contains no real numbers. To Lean, these are different sets, because they have different types. Which one does the symbol \emptyset denote? The answer will be different in different contexts. Lean can often figure out from context which empty set you have in mind, but if it can't, then you have to tell it explicitly by writing $(\emptyset : Set U)$ rather than \emptyset . Fortunately, in our theorems Lean is able to figure out which empty set we have in mind.

With that preparation, we are ready to prove our first preliminary theorem. Since the goal is an equation between sets, our first step is to use the tactic `apply Set.ext`.

Lean File

```
theorem empty_union {U : Type} (B : Set U) :
  ∅ ∪ B = B := by
  apply Set.ext
  done
```

Tactic State in Infoview

▼ case h U : Type B : Set U $\vdash \forall (x : U),$ $x \in \emptyset \cup B \leftrightarrow x \in B$

Based on the form of the goal, our next two tactics should be `fix x : U` and `apply Iff.intro`. This leaves us with two goals, corresponding to the two directions of the biconditional, but we'll focus first on just the left-to-right direction.

Lean File

```
theorem empty_union {U : Type} (B : Set U) :
  ∅ ∪ B = B := by
  apply Set.ext
  fix x : U
  apply Iff.intro
  · -- (→)
    done
  · -- (←)
    done
done
```

Tactic State in Infoview

```
▼ case h.mp
U : Type
B : Set U
x : U
↑ x ∈ ∅ ∪ B → x ∈ B
```

Of course, our next step is to assume $x \in \emptyset \cup B$. To help us see how to move forward, we also write out the definition of this assumption.

Lean File

```
theorem empty_union {U : Type} (B : Set U) :
  ∅ ∪ B = B := by
  apply Set.ext
  fix x : U
  apply Iff.intro
  · -- (→)
    assume h1 : x ∈ ∅ ∪ B
    define at h1
    done
  · -- (←)
    done
done
```

Tactic State in Infoview

```
▼ case h.mp
U : Type
B : Set U
x : U
h1 : x ∈ ∅ ∨ x ∈ B
↑ x ∈ B
```

Now you should see a way to complete the proof: the statement $x \in \emptyset$ is false, so we should be able to apply the disjunctive syllogism rule to $h1$ to infer the goal $x \in B$. To carry out this plan, we'll first have to prove $x \notin \emptyset$. We'll use the `have` tactic, and since there's no obvious term-mode proof to justify it, we'll try a tactic-mode proof.

Lean File

```
theorem empty_union {U : Type} (B : Set U) :
  ∅ ∪ B = B := by
  apply Set.ext
  fix x : U
  apply Iff.intro
  · -- (→)
    assume h1 : x ∈ ∅ ∪ B
    define at h1
    have h2 : x ∉ ∅ := by
      done
      done
    · -- (←)
      done
      done
```

Tactic State in Infoview

U	: Type
B	: Set U
x	: U
h1	: x ∈ ∅ ∨ x ∈ B
	↑ x ∉ ∅

The goal for our “proof within a proof” is a negative statement, so proof by contradiction seems like a good start.

Lean File

```
theorem empty_union {U : Type} (B : Set U) :
  ∅ ∪ B = B := by
  apply Set.ext
  fix x : U
  apply Iff.intro
  · -- (→)
    assume h1 : x ∈ ∅ ∪ B
    define at h1
    have h2 : x ∉ ∅ := by
      by_contra h3
      done
      done
    · -- (←)
      done
      done
```

Tactic State in Infoview

U	: Type
B	: Set U
x	: U
h1	: x ∈ ∅ ∨ x ∈ B
h3	: x ∈ ∅
	↑ False

To see how to use the new assumption `h3`, we use the tactic `define at h3`. The definition Lean gives for the statement $x \in \emptyset$ is `False`. In other words, Lean knows that, by the definition of \emptyset , the statement $x \in \emptyset$ is false. Since `False` is our goal, this completes the inner proof, and we

can return to the main proof.

Lean File

```
theorem empty_union {U : Type} (B : Set U) :
  ∅ ∪ B = B := by
  apply Set.ext
  fix x : U
  apply Iff.intro
  · -- (→)
    assume h1 : x ∈ ∅ ∪ B
    define at h1
    have h2 : x ∉ ∅ := by
      by_contra h3
      define at h3 --h3 : False
      show False from h3
      done
    done
  · -- (←)
    done
  done
```

Tactic State in Infoview

```
▼ case h.mp
U : Type
B : Set U
x : U
h1 : x ∈ ∅ ∨ x ∈ B
h2 : x ∉ ∅
↑ x ∈ B
```

Now that we have established the claim $h2 : x \notin \emptyset$, we can apply the disjunctive syllogism rule to $h1$ and $h2$ to reach the goal. This completes the left-to-right direction of the biconditional proof, so we move on to the right-to-left direction.

Lean File

```

theorem empty_union {U : Type} (B : Set U) :
   $\emptyset \cup B = B$  := by
  apply Set.ext
  fix x : U
  apply Iff.intro
  · -- ( $\rightarrow$ )
    assume h1 : x ∈  $\emptyset \cup B$ 
    define at h1
    have h2 : x ∉  $\emptyset$  := by
      by_contra h3
      define at h3 --h3 : False
      show False from h3
      done
    disj_syll h1 h2 --h1 : x ∈ B
    show x ∈ B from h1
    done
  · -- ( $\leftarrow$ )
    done
  done

```

Tactic State in Infoview

```

▼ case h.mpr
U : Type
B : Set U
x : U
↑ x ∈ B → x ∈  $\emptyset \cup B$ 

```

This direction of the biconditional proof is easier: once we introduce the assumption $h1 : x \in B$, our goal will be $x \in \emptyset \cup B$, which means $x \in \emptyset \vee x \in B$, and we can prove it with the proof tactic `Or.inr h1`.

Lean File

```
theorem empty_union {U : Type} (B : Set U) :
  ∅ U B = B := by
  apply Set.ext
  fix x : U
  apply Iff.intro
  · -- (→)
    assume h1 : x ∈ ∅ U B
    define at h1
    have h2 : x ∉ ∅ := by
      by_contra h3
      define at h3 --h3 : False
      show False from h3
      done
    disj_syll h1 h2 --h1 : x ∈ B
    show x ∈ B from h1
    done
  · -- (←)
    assume h1 : x ∈ B
    show x ∈ ∅ U B from Or.inr h1
    done
done
```

Tactic State in Infoview

No goals

The second fact that was called “clear” in the proof from Example 3.6.2 was the equation $C \cup D = D \cup C$. This looks like an instance of the commutativity of the union operator. Let’s prove that union is commutative.

Lean File

```
theorem union_comm {U : Type} (X Y : Set U) :
  X U Y = Y U X := by
  done
```

Tactic State in Infoview

U : Type
X Y : Set U
 $\vdash X \cup Y = Y \cup X$

Once again, we begin with `apply Set.ext`, which converts the goal to $\forall (x : U), x \in X \cup Y \leftrightarrow x \in Y \cup X$, and then `fix x : U`.

Lean File

```
theorem union_comm {U : Type} (X Y : Set U) :
  X U Y = Y U X := by
  apply Set.ext
  fix x : U
  done
```

Tactic State in Infoview

▼case h
U : Type
X Y : Set U
x : U
 $\vdash x \in X \cup Y \leftrightarrow x \in Y \cup X$

3.6. Existence and Uniqueness Proofs

To understand the goal better, we'll write out the definitions of the two sides of the biconditional. We use an extension of the `define` tactic that allows us to write out the definition of just a part of a given or the goal. The tactic `define : x ∈ X ∪ Y` will replace `x ∈ X ∪ Y` with its definition wherever it appears in the goal, and then `define : x ∈ Y ∪ X` will replace `x ∈ Y ∪ X` with its definition. (Note that `define : X ∪ Y` produces a result that is not as useful. It is usually best to define a complete statement rather than just a part of a statement. As usual, you can add `at` to do the replacements in a given rather than the goal.)

Lean File

```
theorem union_comm {U : Type} (X Y : Set U) :
  X ∪ Y = Y ∪ X := by
  apply Set.ext
  fix x : U
  define : x ∈ X ∪ Y
  define : x ∈ Y ∪ X
  done
```

Tactic State in Infoview

```
▼ case h
U : Type
X Y : Set U
x : U
↑ x ∈ X ∨ x ∈ Y ↔
  x ∈ Y ∨ x ∈ X
```

By the way, there are similar extensions of all of the tactics `contrapos`, `demorgan`, `conditional`, `double_neg`, `bicond_neg`, and `quant_neg` that allow you to use a logical equivalence to rewrite just a part of a formula. For example, if your goal is $P \wedge (\neg Q \rightarrow R)$, then the tactic `contrapos : \neg Q \rightarrow R` will change the goal to $P \wedge (\neg R \rightarrow Q)$. If you have a given $h : P \rightarrow \forall (x : U), Q x$, then the tactic `quant_neg : \neg \forall (x : U), Q x at h` will change h to $h : P \rightarrow \exists (x : U), \neg Q x$.

Returning to our proof of `union_comm`: the goal is now $x \in X \vee x \in Y \leftrightarrow x \in Y \vee x \in X$. You could prove this by a somewhat tedious application of the rules for biconditionals and disjunctions that were discussed in the last two sections, and we invite you to try it. But there is another possibility. The goal now has the form $P \vee Q \leftrightarrow Q \vee P$, which is the commutative law for “or” (see Section 1.2 of *HTPI*). We saw in a previous example that Lean has, in its library, the associative law for “and”; it is called `and_assoc`. Does Lean also know the commutative law for “or”?

Try typing `#check @or_` in VS Code. After a few seconds, a pop-up window appears with possible completions of this command. You will see `or_assoc` on the list, as well as `or_comm`. Select `or_comm`, and you'll get this response: `@or_comm : ∀ {a b : Prop}, a ∨ b ↔ b ∨ a`. Since `a` and `b` are implicit arguments in this theorem, you can use `or_comm` to prove any statement of the form `a ∨ b ↔ b ∨ a`, where Lean will figure out for itself what `a` and `b` stand for. In particular, `or_comm` will prove our current goal.

3.6. Existence and Uniqueness Proofs

Lean File

```
theorem union_comm {U : Type} (X Y : Set U) :
  X ∪ Y = Y ∪ X := by
  apply Set.ext
  fix x : U
  define : x ∈ X ∪ Y
  define : x ∈ Y ∪ X
  show x ∈ X ∨ x ∈ Y ↔ x ∈ Y ∨ x ∈ X from or_comm
  done
```

Tactic State in Infoview

No goals

We have now proven the two statements that were said to be “clearly” true in the proof in Example 3.6.2 of *HTPI*, and we have given them names. And that means that we can now use these theorems, in the file containing these proofs, to prove other theorems. As with any theorem in Lean’s library, you can use the `#check` command to confirm what these theorems say. If you type `#check @empty_union` and `#check @union_comm`, you will get these results:

```
@empty_union : ∀ {U : Type} (B : Set U), ∅ ∪ B = B
```

```
@union_comm : ∀ {U : Type} (X Y : Set U), X ∪ Y = Y ∪ X
```

Notice that in both theorems we used curly braces when we introduced the type `U`, so it is an implicit argument and will not need to be specified when we apply the theorems. (Why did we decide to make `U` an implicit argument? Well, when we apply the theorem `empty_union` we will be specifying the set `B`, and when we apply `union_comm` we will be specifying the sets `X` and `Y`. Lean can figure out what `U` is by examining the types of these sets, so there is no need to specify it separately.)

We are finally ready to prove the theorem from Example 3.6.2. Here is the theorem:

Lean File

```
theorem Example_3_6_2 (U : Type) :
  ∃! (A : Set U), ∀ (B : Set U),
  A ∪ B = B := by
  done
```

Tactic State in Infoview

U : Type
 $\vdash \exists! (A : Set U),$
 $\forall (B : Set U),$
 $A \cup B = B$

The goal starts with `∃!`, so we use our new tactic, `exists_unique`.

3.6. Existence and Uniqueness Proofs

Lean File

```
theorem Example_3_6_2 (U : Type) :
  ∃! (A : Set U), ∀ (B : Set U),
  A ∪ B = B := by
  exists_unique
  done
```

Tactic State in Infoview

```
▼ case Existence
U : Type
↑ ∃ (A : Set U),
  ∀ (B : Set U),
  A ∪ B = B
▼ case Uniqueness
U : Type
↑ ∀ (A_1 A_2 : Set U),
  (∀ (B : Set U),
    A_1 ∪ B = B) →
  (∀ (B : Set U),
    A_2 ∪ B = B) →
  A_1 = A_2
```

We have two goals, labeled **Existence** and **Uniqueness**. Imitating the proof from *HTPI*, we prove existence by using the value \emptyset for A .

Lean File

```
theorem Example_3_6_2 (U : Type) :
  ∃! (A : Set U), ∀ (B : Set U),
  A ∪ B = B := by
  exists_unique
  . -- Existence
    apply Exists.intro ∅
    done
  . -- Uniqueness
  done
done
```

Tactic State in Infoview

```
▼ case Existence
U : Type
↑ ∀ (B : Set U),
  ∅ ∪ B = B
```

The goal is now precisely the statement of the theorem `empty_union`, so we can prove it by simply citing that theorem.

3.6. Existence and Uniqueness Proofs

Lean File

```
theorem Example_3_6_2 (U : Type) :
  ∃! (A : Set U), ∀ (B : Set U),
  A ∪ B = B := by
exists_unique
· -- Existence
  apply Exists.intro ⊥
  show ∀ (B : Set U), ∅ ∪ B = B from empty_union
  done
· -- Uniqueness

done
done
```

Tactic State in Infoview

```
▼ case Uniqueness
U : Type
↑ ∀ (A_1 A_2 : Set U),
  (∀ (B : Set U),
    A_1 ∪ B = B) →
  (∀ (B : Set U),
    A_2 ∪ B = B) →
  A_1 = A_2
```

For the uniqueness proof, we begin by introducing arbitrary sets C and D and assuming $\forall (B : \text{Set } U)$, $C \cup B = B$ and $\forall (B : \text{Set } U)$, $D \cup B = B$, exactly as in the *HTPI* proof.

Lean File

```
theorem Example_3_6_2 (U : Type) :
  ∃! (A : Set U), ∀ (B : Set U),
  A ∪ B = B := by
exists_unique
· -- Existence
  apply Exists.intro ⊥
  show ∀ (B : Set U), ∅ ∪ B = B from empty_union
  done
· -- Uniqueness
  fix C : Set U; fix D : Set U
  assume h1 : ∀ (B : Set U), C ∪ B = B
  assume h2 : ∀ (B : Set U), D ∪ B = B
  done
done
```

Tactic State in Infoview

```
▼ case Uniqueness
U : Type
C D : Set U
h1 : ∀ (B : Set U),
  C ∪ B = B
h2 : ∀ (B : Set U),
  D ∪ B = B
↑ C = D
```

The next step in *HTPI* was to apply $h1$ to D , and $h2$ to C . We do the same thing in Lean.

Lean File

```
theorem Example_3_6_2 (U : Type) :
  ∃! (A : Set U), ∀ (B : Set U),
  A ∪ B = B := by
exists_unique
· -- Existence
  apply Exists.intro ⊥
  show ∀ (B : Set U), ⊥ ∪ B = B from empty_union
  done
· -- Uniqueness
  fix C : Set U; fix D : Set U
  assume h1 : ∀ (B : Set U), C ∪ B = B
  assume h2 : ∀ (B : Set U), D ∪ B = B
  have h3 : C ∪ D = D := h1 D
  have h4 : D ∪ C = C := h2 C
  done
done
```

Tactic State in Infoview

```
▼ case Uniqueness
U : Type
C D : Set U
h1 : ∀ (B : Set U),
C ∪ B = B
h2 : ∀ (B : Set U),
D ∪ B = B
h3 : C ∪ D = D
h4 : D ∪ C = C
↑ C = D
```

The goal can now be achieved by stringing together a sequence of equations: $C = D \cup C = C \cup D = D$. The first of these equations is `h4.symm`—that is, `h4` read backwards; the second follows from the commutative law for union; and the third is `h3`. We saw in Section 3.4 that you can prove a biconditional statement in Lean by stringing together a sequence of biconditionals in a calculational proof. Exactly the same method applies to equations. Here is the complete proof of the theorem:

```
theorem Example_3_6_2 (U : Type) :
  ∃! (A : Set U), ∀ (B : Set U),
  A ∪ B = B := by
exists_unique
· -- Existence
  apply Exists.intro ⊥
  show ∀ (B : Set U), ⊥ ∪ B = B from empty_union
  done
· -- Uniqueness
  fix C : Set U; fix D : Set U
  assume h1 : ∀ (B : Set U), C ∪ B = B
  assume h2 : ∀ (B : Set U), D ∪ B = B
  have h3 : C ∪ D = D := h1 D
  have h4 : D ∪ C = C := h2 C
  show C = D from
    calc C
    _ = D ∪ C := h4.symm
```

```
_ = C ∪ D := union_comm D C
_ = D := h3
done
done
```

Since the statement $\exists! (x : U), P x$ asserts both the existence and the uniqueness of an object satisfying the predicate P , we have the following strategy for using a given of this form (*HTPI* p. 159):

To use a given of the form $\exists! (x : U), P x$:

Introduce a new variable, say a , into the proof to stand for an object of type U for which $P a$ is true. You may also assert that $\forall (x_1 x_2 : U), P x_1 \rightarrow P x_2 \rightarrow x_1 = x_2$.

If you have a given $h : \exists! (x : U), P x$, then the tactic `obtain (a : U) (h1 : P a) (h2 : ∀ (x_1 x_2 : U), P x_1 → P x_2 → x_1 = x_2)` from h will introduce into the tactic state a new variable a of type U and new givens $(h1 : P a)$ and $(h2 : \forall (x_1 x_2 : U), P x_1 \rightarrow P x_2 \rightarrow x_1 = x_2)$. To illustrate the use of this tactic, let's prove the theorem in Example 3.6.4 of *HTPI*.

Lean File

```
theorem Example_3_6_4 (U : Type) (A B C : Set U)
  (h1 : ∃ (x : U), x ∈ A ∩ B)
  (h2 : ∃ (x : U), x ∈ A ∩ C)
  (h3 : ∃! (x : U), x ∈ A) :
  ∃ (x : U), x ∈ B ∩ C := by
done
```

Tactic State in Infoview

U : Type
A B C : Set U
h1 : ∃ (x : U), x ∈ A ∩ B
h2 : ∃ (x : U), x ∈ A ∩ C
h3 : ∃! (x : U), x ∈ A ⋮ ∃ (x : U), x ∈ B ∩ C

We begin by applying the `obtain` tactic to $h1$, $h2$, and $h3$. In the case of $h3$, we get an extra given asserting the uniqueness of the element of A . We also write out the definitions of two of the new givens we obtain.

3.6. Existence and Uniqueness Proofs

Lean File

```
theorem Example_3_6_4 (U : Type) (A B C : Set U)
  (h1 : ∃ (x : U), x ∈ A ∩ B)
  (h2 : ∃ (x : U), x ∈ A ∩ C)
  (h3 : ∃! (x : U), x ∈ A) :
  ∃ (x : U), x ∈ B ∩ C := by
  obtain (b : U) (h4 : b ∈ A ∩ B) from h1
  obtain (c : U) (h5 : c ∈ A ∩ C) from h2
  obtain (a : U) (h6 : a ∈ A) (h7 : ∀ (y z : U),
    y ∈ A → z ∈ A → y = z) from h3
  define at h4; define at h5
  done
```

Tactic State in Infoview

```
U : Type
A B C : Set U
h1 : ∃ (x : U),
  x ∈ A ∩ B
h2 : ∃ (x : U),
  x ∈ A ∩ C
h3 : ∃! (x : U), x ∈ A
b : U
h4 : b ∈ A ∧ b ∈ B
c : U
h5 : c ∈ A ∧ c ∈ C
a : U
h6 : a ∈ A
h7 : ∀ (y z : U),
  y ∈ A → z ∈ A → y = z
  ⊢ ∃ (x : U), x ∈ B ∩ C
```

The key to the rest of the proof is the observation that, by the uniqueness of the element of A , b must be equal to c . To justify this conclusion, note that by two applications of universal instantiation, $h7\ b\ c$ is a proof of $b \in A \rightarrow c \in A \rightarrow b = c$, and therefore by two applications of modus ponens, $h7\ b\ c\ h4.\text{left}\ h5.\text{left}$ is a proof of $b = c$.

Lean File

```
theorem Example_3_6_4 (U : Type) (A B C : Set U)
  (h1 : ∃ (x : U), x ∈ A ∩ B)
  (h2 : ∃ (x : U), x ∈ A ∩ C)
  (h3 : ∃! (x : U), x ∈ A) :
  ∃ (x : U), x ∈ B ∩ C := by
  obtain (b : U) (h4 : b ∈ A ∩ B) from h1
  obtain (c : U) (h5 : c ∈ A ∩ C) from h2
  obtain (a : U) (h6 : a ∈ A) (h7 : ∀ (y z : U),
    y ∈ A → z ∈ A → y = z) from h3
  define at h4; define at h5
  have h8 : b = c := h7\ b\ c\ h4.\text{left}\ h5.\text{left}
  done
```

Tactic State in Infoview

```
U : Type
A B C : Set U
h1 : ∃ (x : U),
  x ∈ A ∩ B
h2 : ∃ (x : U),
  x ∈ A ∩ C
h3 : ∃! (x : U), x ∈ A
b : U
h4 : b ∈ A ∧ b ∈ B
c : U
h5 : c ∈ A ∧ c ∈ C
a : U
h6 : a ∈ A
h7 : ∀ (y z : U),
  y ∈ A → z ∈ A → y = z
h8 : b = c
  ⊢ ∃ (x : U), x ∈ B ∩ C
```

3.6. Existence and Uniqueness Proofs

For our next step, we will need a new tactic. Since we have $h8 : b = c$, we should be able to replace b with c anywhere it appears. The tactic that allows us to do this called `rewrite`. If h is a proof of any equation $s = t$, then `rewrite [h]` will replace all occurrences of s in the goal with t . Notice that it is the left side of the equation that is replaced with the right side; if you want the replacement to go in the other direction, so that t is replaced with s , you can use `rewrite [←h]`. (Alternatively, since $h.\text{symm}$ is a proof of $t = s$, you can use `rewrite [h.\text{symm}]`.) You can also apply the `rewrite` tactic to biconditional statements. If you have $h : P \leftrightarrow Q$, then `rewrite [h]` will cause all occurrences of P in the goal to be replaced with Q (and `rewrite [←h]` will replace Q with P).

As with many other tactics, you can add `at h'` to specify that the replacement should be done in the given h' rather than the goal. In our case, `rewrite [h8] at h4` will change both occurrences of b in $h4$ to c .

Lean File

```
theorem Example_3_6_4 (U : Type) (A B C : Set U)
  (h1 : ∃ (x : U), x ∈ A ∩ B)
  (h2 : ∃ (x : U), x ∈ A ∩ C)
  (h3 : ∃! (x : U), x ∈ A) :
  ∃ (x : U), x ∈ B ∩ C := by
  obtain (b : U) (h4 : b ∈ A ∩ B) from h1
  obtain (c : U) (h5 : c ∈ A ∩ C) from h2
  obtain (a : U) (h6 : a ∈ A) (h7 : ∀ (y z : U),
    y ∈ A → z ∈ A → y = z) from h3
  define at h4; define at h5
  have h8 : b = c := h7 b c h4.left h5.left
  rewrite [h8] at h4
  done
```

Tactic State in Infoview

U : Type
A B C : Set U
h1 : ∃ (x : U), x ∈ A ∩ B
h2 : ∃ (x : U), x ∈ A ∩ C
h3 : ∃! (x : U), x ∈ A
b c : U
h4 : c ∈ A ∧ c ∈ B
h5 : c ∈ A ∧ c ∈ C
a : U
h6 : a ∈ A
h7 : ∀ (y z : U), y ∈ A → z ∈ A → y = z
h8 : b = c
↑ ∃ (x : U), x ∈ B ∩ C

Now the right sides of $h4$ and $h5$ tell us that we can prove the goal by plugging in c for x . Here is the complete proof:

```
theorem Example_3_6_4 (U : Type) (A B C : Set U)
  (h1 : ∃ (x : U), x ∈ A ∩ B)
  (h2 : ∃ (x : U), x ∈ A ∩ C)
  (h3 : ∃! (x : U), x ∈ A) :
  ∃ (x : U), x ∈ B ∩ C := by
  obtain (b : U) (h4 : b ∈ A ∩ B) from h1
  obtain (c : U) (h5 : c ∈ A ∩ C) from h2
  obtain (a : U) (h6 : a ∈ A) (h7 : ∀ (y z : U),
```

3.6. Existence and Uniqueness Proofs

```

y ∈ A → z ∈ A → y = z) from h3
define at h4; define at h5
have h8 : b = c := h7 b c h4.left h5.left
rewrite [h8] at h4
show ∃ (x : U), x ∈ B ∩ C from
  Exists.intro c (And.intro h4.right h5.right)
done

```

You might want to compare the Lean proof above to the proof of this theorem as it appears in *HTPI* (*HTPI* p. 160):

Theorem. Suppose A , B , and C are sets, A and B are not disjoint, A and C are not disjoint, and A has exactly one element. Then B and C are not disjoint

Proof. Since A and B are not disjoint, we can let b be something such that $b \in A$ and $b \in B$. Similarly, since A and C are not disjoint, there is some object c such that $c \in A$ and $c \in C$. Since A has only one element, we must have $b = c$. Thus $b = c \in B \cap C$ and therefore B and C are not disjoint. \square

Before ending this section, we return to the question of how you can tell if a theorem you want to use is in Lean’s library. In an earlier example, we guessed that the commutative law for “or” might be in Lean’s library, and we were then able to use the `#check` command to confirm it. But there is another technique that we could have used: the tactic `apply?`, which asks Lean to search through its library of theorems to see if there is one that could be applied to prove the goal. Let’s return to our proof of the theorem `union_comm`, which started like this:

Lean File

```

theorem union_comm {U : Type} (X Y : Set U) :
  X ∪ Y = Y ∪ X := by
  apply Set.ext
  fix x : U
  define : x ∈ X ∪ Y
  define : x ∈ Y ∪ X
  done

```

Tactic State in Infoview

```

▼ case h
U : Type
X Y : Set U
x : U
↑ x ∈ X ∨ x ∈ Y ↔
  x ∈ Y ∨ x ∈ X

```

Now let’s give the `apply?` tactic a try.

```

theorem union_comm {U : Type} (X Y : Set U) :
  X ∪ Y = Y ∪ X := by
  apply Set.ext
  fix x : U

```

```
define : x ∈ X ∪ Y
define : x ∈ Y ∪ X
apply?
done
```

It takes a few seconds for Lean to search its library of theorems, but eventually a blue squiggle appears under `apply?`, indicating that the tactic has produced an answer. You will find the answer in the Infoview pane: Try this: `exact Or.comm`. The word `exact` is the name of a tactic that we have not discussed; it is a shorthand for `show _ from`, where the blank gets filled in with the goal. Thus, you can think of `apply?`'s answer as a shortened form of the tactic

```
show x ∈ X ∨ x ∈ Y ↔ x ∈ Y ∨ x ∈ X from Or.comm
```

The command `#check @Or.comm` will tell you that `Or.comm` is just an alternative name for the theorem `or_comm`. So the step suggested by the `apply?` tactic is essentially the same as the step we used earlier to complete the proof.

Usually your proof will be more readable if you use the `show` tactic to state explicitly the goal that is being proven. This also gives Lean a chance to correct you if you have become confused about what goal you are proving. But sometimes—for example, if the goal is very long—it is convenient to use the `exact` tactic instead. You might think of `exact` as meaning “the following is a term-mode proof that is exactly what is needed to prove the goal.”

The `apply?` tactic has not only come up with a suggested tactic, it has applied that tactic, and the proof is now complete. You can confirm that the tactic completes the proof by replacing the line `apply?` in the proof with `apply?`'s suggested `exact` tactic.

The `apply?` tactic is somewhat unpredictable; sometimes it is able to find the right theorem in the library, and sometimes it isn't. But it is always worth a try. There are also tools available on the internet for searching Lean's library, including [LeanSearch](#), [Moogle](#), and [Google](#). Another way to try to find theorems is to visit the documentation page for Lean's mathematics library, which can be found at https://leanprover-community.github.io/mathlib4_docs/.

Exercises

1. `theorem Exercise_3_4_15 (U : Type) (B : Set U) (F : Set (Set U)) :`

$$\begin{aligned} & \cup_{\emptyset} \{X : Set U \mid \exists (A : Set U), A \in F \wedge X = A \setminus B\} \\ & \subseteq \cup_{\emptyset} (F \setminus \mathcal{P} B) := \text{sorry} \end{aligned}$$

```

2. theorem Exercise_3_5_9 (U : Type) (A B : Set U)
  (h1 : ℙ(A ∪ B) = ℙ A ∪ ℙ B) : A ⊆ B ∨ B ⊆ A := by
  --Hint: Start like this:
  have h2 : A ∪ B ∈ ℙ(A ∪ B) := sorry
  sorry

3. theorem Exercise_3_6_6b (U : Type) :
  ∃! (A : Set U), ∀ (B : Set U), A ∪ B = A := sorry

4. theorem Exercise_3_6_7b (U : Type) :
  ∃! (A : Set U), ∀ (B : Set U), A ∩ B = A := sorry

5. theorem Exercise_3_6_8a (U : Type) : ∀ (A : Set U),
  ∃! (B : Set U), ∀ (C : Set U), C \ A = C ∩ B := sorry

6. theorem Exercise_3_6_10 (U : Type) (A : Set U)
  (h1 : ∀ (F : Set (Set U)), U₀ F = A → A ∈ F) :
  ∃! (x : U), x ∈ A := by
  --Hint: Start like this:
  set F₀ : Set (Set U) := {X : Set U | X ⊆ A ∧ ∃! (x : U), x ∈ X}
  --Now F₀ is in the tactic state, with the definition above
  have h2 : U₀ F₀ = A := sorry
  sorry

```

3.7. More Examples of Proofs

It is finally time to discuss proofs involving algebraic reasoning. Lean has types for several different kinds of numbers. `Nat` is the type of natural numbers—that is, the numbers 0, 1, 2, `Int` is the type of integers, `Rat` is the type of rational numbers, `Real` is the type of real numbers, and `Complex` is the type of complex numbers. Lean also uses the notation `N`, `Z`, `Q`, `R`, and `C` for these types. (If you want to use those names for the number types, you can enter them by typing `\N`, `\Z`, `\Q`, `\R`, and `\C`.) To write formulas involving arithmetic operations, you should use the symbols `+` for addition, `-` for subtraction, `*` for multiplication, `/` for division, and `^` for exponentiation. You can enter the symbols `≤`, `≥`, and `≠` by typing `\le`, `\ge`, and `\ne`, respectively. We will discuss some of the more subtle points of algebraic reasoning in Chapter 6. For the moment, you are best off avoiding subtraction and division when working with natural numbers and avoiding division when working with integers.

To see what's involved in proving theorems about numbers in Lean, we'll turn to a few examples from earlier in Chapter 3 of *HTPI*. We begin with Theorem 3.3.7, which concerns divisibility of integers. As in *HTPI*, for integers x and y , we will write $x \mid y$ to mean that x divides y , or y is divisible by x . The formal definition is that $x \mid y$ means that there is an integer k such

3.7. More Examples of Proofs

that $y = x * k$. For example, $3 \mid 12$, since $12 = 3 * 4$. Lean knows this notation, but there is an important warning: to type the vertical line that means “divides,” you must type `\|`, not simply `|`. (There are two slightly different vertical line symbols, and you have to look closely to see that they are different: `|` and `\|`. It is the second one that means “divides” in Lean, and to enter it you must type `\|`.) Here is Theorem 3.3.7, written using our usual rephrasing of a statement of the form $A \wedge B \rightarrow C$ as $A \rightarrow B \rightarrow C$.

Lean File

```
theorem Theorem_3_3_7 :  
  ∀ (a b c : Int), a | b → b | c → a | c := by  
  
done
```

Tactic State in Infoview

```
↑ ∀ (a b c : ℤ),  
  a | b → b | c → a | c
```

Of course, we begin the proof by introducing arbitrary integers a , b , and c , and assuming $a \mid b$ and $b \mid c$. We also write out the definitions of our assumptions and the goal.

Lean File

```
theorem Theorem_3_3_7 :  
  ∀ (a b c : Int), a | b → b | c → a | c := by  
  fix a : Int; fix b : Int; fix c : Int  
  assume h1 : a | b; assume h2 : b | c  
  define at h1; define at h2; define  
done
```

Tactic State in Infoview

```
a b c : ℤ  
h1 : ∃ (c : ℤ),  
  b = a * c  
h2 : ∃ (c_1 : ℤ),  
  c = b * c_1  
↑ ∃ (c_1 : ℤ),  
  c = a * c_1
```

We always use existential givens right away, so we use $h1$ and $h2$ to introduce two new variables, m and n .

Lean File

```
theorem Theorem_3_3_7 :  
  ∀ (a b c : Int), a | b → b | c → a | c := by  
  fix a : Int; fix b : Int; fix c : Int  
  assume h1 : a | b; assume h2 : b | c  
  define at h1; define at h2; define  
  obtain (m : Int) (h3 : b = a * m) from h1  
  obtain (n : Int) (h4 : c = b * n) from h2  
done
```

Tactic State in Infoview

```
a b c : ℤ  
h1 : ∃ (c : ℤ),  
  b = a * c  
h2 : ∃ (c_1 : ℤ),  
  c = b * c_1  
m : ℤ  
h3 : b = a * m  
n : ℤ  
h4 : c = b * n  
↑ ∃ (c_1 : ℤ),  
  c = a * c_1
```

If we substitute the value for b given in $h3$ into $h4$, we will see how to reach the goal. Of course, the `rewrite` tactic is what we need for this.

Lean File

```
theorem Theorem_3_3_7 :  
  ∀ (a b c : Int), a | b → b | c → a | c := by  
  fix a : Int; fix b : Int; fix c : Int  
  assume h1 : a | b; assume h2 : b | c  
  define at h1; define at h2; define  
  obtain (m : Int) (h3 : b = a * m) from h1  
  obtain (n : Int) (h4 : c = b * n) from h2  
  rewrite [h3] at h4 --h4 : c = a * m * n  
  done
```

Tactic State in Infoview

```
a b c : ℤ  
h1 : ∃ (c : ℤ),  
  b = a * c  
h2 : ∃ (c_1 : ℤ),  
  c = b * c_1  
m : ℤ  
h3 : b = a * m  
n : ℤ  
h4 : c = a * m * n  
  ⊢ ∃ (c_1 : ℤ),  
    c = a * c_1
```

Looking at h4, we see that the value we should use for c_1 in the goal is $m * n$.

Lean File

```
theorem Theorem_3_3_7 :  
  ∀ (a b c : Int), a | b → b | c → a | c := by  
  fix a : Int; fix b : Int; fix c : Int  
  assume h1 : a | b; assume h2 : b | c  
  define at h1; define at h2; define  
  obtain (m : Int) (h3 : b = a * m) from h1  
  obtain (n : Int) (h4 : c = b * n) from h2  
  rewrite [h3] at h4 --h4 : c = a * m * n  
  apply Exists.intro (m * n)  
  done
```

Tactic State in Infoview

```
a b c : ℤ  
h1 : ∃ (c : ℤ),  
  b = a * c  
h2 : ∃ (c_1 : ℤ),  
  c = b * c_1  
m : ℤ  
h3 : b = a * m  
n : ℤ  
h4 : c = a * m * n  
  ⊢ c = a * (m * n)
```

Note that in the application of `Exists.intro`, the parentheses around $m * n$ are necessary to help Lean parse the line correctly. Comparing h4 to the goal, you might think that we can finish the proof with `show c = a * (m * n) from h4`. But if you try it, you will get an error message. What's the problem? The difference in the parentheses is the clue. Lean groups the arithmetic operations $+$, $-$, $*$, and $/$ to the left, so h4 means $h4 : c = (a * m) * n$, which is not quite the same as the goal. To prove the goal, we will need to apply the associative law for multiplication.

We have already seen that `and_assoc` is Lean's name for the associative law for “and”. Perhaps you can guess that the name for the associative law for multiplication is `mul_assoc`. If you type `#check @mul_assoc`, Lean's response will be:

```
@mul_assoc : ∀ {G : Type u_1} [inst : Semigroup G] (a b c : G),  
  a * b * c = a * (b * c)
```

3.7. More Examples of Proofs

The implicit arguments in this cases are a little complicated (the expression `[inst : Semigroup G]` represents yet another kind of implicit argument). But what they mean is that `mul_assoc` can be used to prove any statement of the form $\forall (a b c : G), a * b * c = a * (b * c)$, as long as G is a type that has an associative multiplication operation. In particular, `mul_assoc` can be used as a proof of $\forall (a b c : \text{Int}), a * b * c = a * (b * c)$. (There are also versions of this theorem for particular number types. You can use the `#check` command to verify the theorems `Nat.mul_assoc : $\forall (a b c : \mathbb{N}), a * b * c = a * (b * c)$` , `Int.mul_assoc : $\forall (a b c : \mathbb{Z}), a * b * c = a * (b * c)$` , and so on.)

Returning to our proof of Theorem 3.3.7, by three applications of universal instantiation, `mul_assoc a m n` is a proof of $a * m * n = a * (m * n)$, and that is exactly what we need to finish the proof. The tactic `rewrite [mul_assoc a m n]` at `h4` will replace $a * m * n$ in `h4` with $a * (m * n)$.

Lean File

```
theorem Theorem_3_3_7 :
   $\forall (a b c : \text{Int}), a | b \rightarrow b | c \rightarrow a | c := \text{by}$ 
  fix a : Int; fix b : Int; fix c : Int
  assume h1 : a | b; assume h2 : b | c
  define at h1; define at h2; define
  obtain (m : Int) (h3 : b = a * m) from h1
  obtain (n : Int) (h4 : c = b * n) from h2
  rewrite [h3] at h4 --h4 : c = a * m * n
  apply Exists.intro (m * n)
  rewrite [mul_assoc a m n] at h4
  done
```

Tactic State in Infoview

a b c : \mathbb{Z}
h1 : $\exists (c : \mathbb{Z}),$
$b = a * c$
h2 : $\exists (c_1 : \mathbb{Z}),$
$c = b * c_1$
m : \mathbb{Z}
h3 : $b = a * m$
n : \mathbb{Z}
h4 : $c = a * (m * n)$
$\vdash c = a * (m * n)$

By the way, this is a case in which Lean could have figured out some details on its own. If we had used `rewrite [mul_assoc _ _ _]` at `h4`, then Lean would have figured out that the blanks had to be filled in with a , m , and n . And as with the `apply` tactic, blanks at the end of `rewrite` rules can be left out, so even `rewrite [mul_assoc]` at `h4` would have worked.

Of course, now `h4` really does match the goal exactly, so we can use it to complete the proof.

```
theorem Theorem_3_3_7 :
   $\forall (a b c : \text{Int}), a | b \rightarrow b | c \rightarrow a | c := \text{by}$ 
  fix a : Int; fix b : Int; fix c : Int
  assume h1 : a | b; assume h2 : b | c
  define at h1; define at h2; define
  obtain (m : Int) (h3 : b = a * m) from h1
  obtain (n : Int) (h4 : c = b * n) from h2
  rewrite [h3] at h4 --h4 : c = a * m * n
  apply Exists.intro (m * n)
```

```

rewrite [mul_assoc a m n] at h4
show c = a * (m * n) from h4
done
    
```

As usual, you might find it instructive to compare the Lean proof above to the proof of this theorem in *HTPI*.

For our next example, we'll do a somewhat more complex proof concerning divisibility. Here is the proof from *HTPI* (*HTPI* p. 139).

Theorem 3.4.7. *For every integer n , $6 \mid n$ iff $2 \mid n$ and $3 \mid n$.*

Proof. Let n be an arbitrary integer.

(\rightarrow) Suppose $6 \mid n$. Then we can choose an integer k such that $6k = n$. Therefore $n = 6k = 2(3k)$, so $2 \mid n$, and similarly $n = 6k = 3(2k)$, so $3 \mid n$.

(\leftarrow) Suppose $2 \mid n$ and $3 \mid n$. Then we can choose integers j and k such that $n = 2j$ and $n = 3k$. Therefore $6(j - k) = 6j - 6k = 3(2j) - 2(3k) = 3n - 2n = n$, so $6 \mid n$. \square

Let's try writing the proof in Lean. We use exactly the same strategy as in the *HTPI* proof: we begin by fixing an arbitrary integer n , and then we prove the two directions of the biconditional separately.

Lean File

```

theorem Theorem_3_4_7 :
  ∀ (n : Int), 6 ∣ n ↔ 2 ∣ n ∧ 3 ∣ n := by
  fix n : Int
  apply Iff.intro
  done
    
```

Tactic State in Infoview

```

▼ case mp
n : ℤ
↑ 6 ∣ n → 2 ∣ n ∧ 3 ∣ n
▼ case mpr
n : ℤ
↑ 2 ∣ n ∧ 3 ∣ n → 6 ∣ n
    
```

For the left-to-right direction, we assume $6 \mid n$, and since the definition of this assumption is an existential statement, we immediately apply existential instantiation.

Lean File

```
theorem Theorem_3_4_7 :  
  ∀ (n : Int), 6 | n ↔ 2 | n ∧ 3 | n := by  
  fix n : Int  
  apply Iff.intro  
  . -- (→)  
    assume h1 : 6 | n; define at h1  
    obtain (k : Int) (h2 : n = 6 * k) from h1  
    done  
  . -- (←)  
  
  done  
done
```

Tactic State in Infoview

```
▼ case mp  
n : ℤ  
h1 : ∃ (c : ℤ),  
  n = 6 * c  
k : ℤ  
h2 : n = 6 * k  
  ⊢ 2 | n ∧ 3 | n
```

Our goal is now a conjunction, so we prove the two conjuncts separately. Focusing just on the first one, $2 | n$, we write out the definition to decide how to proceed.

Lean File

```
theorem Theorem_3_4_7 :  
  ∀ (n : Int), 6 | n ↔ 2 | n ∧ 3 | n := by  
  fix n : Int  
  apply Iff.intro  
  . -- (→)  
    assume h1 : 6 | n; define at h1  
    obtain (k : Int) (h2 : n = 6 * k) from h1  
    apply And.intro  
    . -- Proof that 2 | n  
      define  
      done  
    . -- Proof that 3 | n  
  
    done  
  . -- (←)  
  
  done  
done
```

Tactic State in Infoview

```
▼ case mp.left  
n : ℤ  
h1 : ∃ (c : ℤ),  
  n = 6 * c  
k : ℤ  
h2 : n = 6 * k  
  ⊢ ∃ (c : ℤ), n = 2 * c
```

Since we have $n = 6 * k = 2 * 3 * k$, it looks like $3 * k$ is the value we should use for c .

Lean File

```

theorem Theorem_3_4_7 :
  ∀ (n : Int), 6 | n ↔ 2 | n ∧ 3 | n := by
  fix n : Int
  apply Iff.intro
  . -- (→)
    assume h1 : 6 | n; define at h1
    obtain (k : Int) (h2 : n = 6 * k) from h1
    apply And.intro
    . -- Proof that 2 | n
      define
      apply Exists.intro (3 * k)
      done
    . -- Proof that 3 | n

    done
  done
  . -- (←)

  done
done

```

Tactic State in Infoview

```

▼ case mp.left
n : ℤ
h1 : ∃ (c : ℤ),
  n = 6 * c
k : ℤ
h2 : n = 6 * k
↑ n = 2 * (3 * k)

```

Once again, if you think carefully about it, you will see that in order to deduce the goal from $h2$, we will need to use the associativity of multiplication to rewrite the goal as $n = 2 * 3 * k$. As we have already seen, `mul_assoc 2 3 k` is a proof of $2 * 3 * k = 2 * (3 * k)$. Since we want to replace the right side of this equation with the left in the goal, we'll use the tactic `rewrite [←mul_assoc 2 3 k]`.

Lean File

```

theorem Theorem_3_4_7 :
  ∀ (n : Int), 6 | n ↔ 2 | n ∧ 3 | n := by
  fix n : Int
  apply Iff.intro
  . -- (→)
    assume h1 : 6 | n; define at h1
    obtain (k : Int) (h2 : n = 6 * k) from h1
    apply And.intro
    . -- Proof that 2 | n
      define
      apply Exists.intro (3 * k)
      rewrite [←mul_assoc 2 3 k]
      done
    . -- Proof that 3 | n

    done
  done
. -- (←)

done
done

```

Tactic State in Infoview

```

▼ case mp.left
n : ℤ
h1 : ∃ (c : ℤ),
  n = 6 * c
k : ℤ
h2 : n = 6 * k
↑ n = 2 * 3 * k

```

Do we have to convince Lean that $2 * 3 = 6$? No, remember that Lean works out definitions on its own. Lean knows the definition of multiplication, and it knows that, according to that definition, $2 * 3$ is equal to 6. So it regards $n = 6 * k$ and $n = 2 * 3 * k$ as definitionally equal, and therefore it will recognize $h2$ as a proof of the goal.

Lean File

```

theorem Theorem_3_4_7 :
  ∀ (n : Int), 6 | n ↔ 2 | n ∧ 3 | n := by
  fix n : Int
  apply Iff.intro
  . -- (→)
    assume h1 : 6 | n; define at h1
    obtain (k : Int) (h2 : n = 6 * k) from h1
    apply And.intro
    . -- Proof that 2 | n
      define
      apply Exists.intro (3 * k)
      rewrite [←mul_assoc 2 3 k]
      show n = 2 * 3 * k from h2
      done
    . -- Proof that 3 | n
      done
  . -- (←)
    done
  done

```

Tactic State in Infoview

```

▼ case mp.right
n : ℤ
h1 : ∃ (c : ℤ),
n = 6 * c
k : ℤ
h2 : n = 6 * k
↑ 3 | n

```

The proof of the next goal, $3 | n$, is similar, and it completes the left-to-right direction of the biconditional.

Lean File

```

theorem Theorem_3_4_7 :
  ∀ (n : Int), 6 | n ↔ 2 | n ∧ 3 | n := by
  fix n : Int
  apply Iff.intro
  . -- (→)
    assume h1 : 6 | n; define at h1
    obtain (k : Int) (h2 : n = 6 * k) from h1
    apply And.intro
    . -- Proof that 2 | n
      define
      apply Exists.intro (3 * k)
      rewrite [←mul_assoc 2 3 k]
      show n = 2 * 3 * k from h2
      done
    . -- Proof that 3 | n
      define
      apply Exists.intro (2 * k)
      rewrite [←mul_assoc 3 2 k]
      show n = 3 * 2 * k from h2
      done
    done
  . -- (←)
    done
done

```

Tactic State in Infoview

```

▼ case mpr
n : ℤ
↑ 2 | n ∧ 3 | n → 6 | n

```

For the right-to-left direction, we begin by assuming $2 \mid n \wedge 3 \mid n$. We write out the definitions of $2 \mid n$ and $3 \mid n$, and since this gives us two existential givens, we apply existential instantiation twice. To save space, we won't repeat the proof of the first half of the proof in the displays below.

Lean File

```
theorem Theorem_3_4_7 :  
  ∀ (n : Int), 6 | n ↔ 2 | n ∧ 3 | n := by  
  fix n : Int  
  apply Iff.intro  
  · -- (→)  
    ...  
  · -- (←)  
    assume h1 : 2 | n ∧ 3 | n  
    have h2 : 2 | n := h1.left  
    have h3 : 3 | n := h1.right  
    define at h2; define at h3; define  
    obtain (j : Int) (h4 : n = 2 * j) from h2  
    obtain (k : Int) (h5 : n = 3 * k) from h3  
    done  
done
```

Tactic State in Infoview

```
▼ case mpr  
n : ℤ  
h1 : 2 | n ∧ 3 | n  
h2 : ∃ (c : ℤ),  
n = 2 * c  
h3 : ∃ (c : ℤ),  
n = 3 * c  
j : ℤ  
h4 : n = 2 * j  
k : ℤ  
h5 : n = 3 * k  
↑ ∃ (c : ℤ),  
n = 6 * c
```

The next step in the *HTPI* proof is a string of equations that proves $6(j - k) = n$, which establishes that $6 | n$. Let's try to do the same thing in Lean, using a calculational proof:

Lean File

```
theorem Theorem_3_4_7 :
  ∀ (n : Int), 6 | n ↔ 2 | n ∧ 3 | n := by
  fix n : Int
  apply Iff.intro
  · -- (→)
    ...
  · -- (←)
    assume h1 : 2 | n ∧ 3 | n
    have h2 : 2 | n := h1.left
    have h3 : 3 | n := h1.right
    define at h2; define at h3; define
    obtain (j : Int) (h4 : n = 2 * j) from h2
    obtain (k : Int) (h5 : n = 3 * k) from h3
    have h6 : 6 * (j - k) = n :=
      calc 6 * (j - k)
        _ = 6 * j - 6 * k := sorry
        _ = 3 * (2 * j) - 2 * (3 * k) := sorry
        _ = 3 * n - 2 * n := sorry
        _ = (3 - 2) * n := sorry
        _ = n := sorry
    show ∃ (c : Int), n = 6 * c from
      Exists.intro (j - k) h6.symm
    done
  done
```

Tactic State in Infoview

No goals

Sometimes the easiest way to write a calculational proof is to justify each line with `sorry` and then go back and fill in real justifications. Lean has accepted the proof above, so we know that we'll have a complete proof if we can replace each `sorry` with a justification.

To justify the first line of the calculational proof, try replacing `sorry` with `by apply?`. Lean comes up with a justification: `Int.mul_sub 6 j k`. The command `#check @Int.mul_sub` tells us that the theorem `Int.mul_sub` means

```
Int.mul_sub : ∀ (a b c : ℤ), a * (b - c) = a * b - a * c
```

Thus, we can fill in `Int.mul_sub 6 j k` as a proof of the first equation.

It looks like we'll have to use the associativity of multiplication again to prove the second equation, but it will take more than one step. Let's try writing a tactic-mode proof. In the display below, we'll just focus on the calculational proof.

Lean File

```

have h6 : 6 * (j - k) = n :=
calc 6 * (j - k)
  _ = 6 * j - 6 * k := Int.mul_sub 6 j k
  _ = 3 * (2 * j) - 2 * (3 * k) := by
    done
  _ = 3 * n - 2 * n := sorry
  _ = (3 - 2) * n := sorry
  _ = n := sorry

```

Tactic State in Infoview

```

n : ℤ
h1 : 2 | n ∧ 3 | n
h2 : ∃ (c : ℤ),
  n = 2 * c
h3 : ∃ (c : ℤ),
  n = 3 * c
j : ℤ
h4 : n = 2 * j
k : ℤ
h5 : n = 3 * k
↑ 6 * j - 6 * k =
  3 * (2 * j) -
  2 * (3 * k)

```

To justify the second equation, we'll have to use associativity to rewrite both $3 * (2 * j)$ as $3 * 2 * j$ and also $2 * (3 * k)$ as $2 * 3 * k$. So we apply the `rewrite` tactic to both of the proofs `mul_assoc 3 2 j : 3 * 2 * j = 3 * (2 * j)` and `mul_assoc 2 3 k : 2 * 3 * k = 2 * (3 * k)`:

Lean File

```

have h6 : 6 * (j - k) = n :=
calc 6 * (j - k)
  _ = 6 * j - 6 * k := Int.mul_sub 6 j k
  _ = 3 * (2 * j) - 2 * (3 * k) := by
    rewrite [←mul_assoc 3 2 j]
    rewrite [←mul_assoc 2 3 k]
    done
  _ = 3 * n - 2 * n := sorry
  _ = (3 - 2) * n := sorry
  _ = n := sorry

```

Tactic State in Infoview

```

n : ℤ
h1 : 2 | n ∧ 3 | n
h2 : ∃ (c : ℤ),
  n = 2 * c
h3 : ∃ (c : ℤ),
  n = 3 * c
j : ℤ
h4 : n = 2 * j
k : ℤ
h5 : n = 3 * k
↑ 6 * j - 6 * k =
  3 * 2 * j -
  2 * 3 * k

```

To finish off the justification of the second equation, we'll use the theorem `Eq.refl`. The command `#check @Eq.refl` gives the result

```
@Eq.refl : ∀ {α : Sort u_1} (a : α), a = a
```

Ignoring the implicit argument α , this should remind you of the theorem `Iff.refl : ∀ (a : Prop), a ↔ a`. Recall that we were able to use `Iff.refl _` to prove not only any statement of the form $a \leftrightarrow a$, but also statements of the form $a \leftrightarrow a'$, where a and a' are definitionally

equal. Similarly, `Eq.refl _` will prove any equation of the form $a = a'$, where a and a' are definitionally equal. Since Lean knows that, by definition, $3 * 2 = 6$ and $2 * 3 = 6$, the goal has this form. Thus we can complete the proof with the tactic `show 6 * j - 6 * k = 3 * 2 * j - 2 * 3 * k from Eq.refl _`. As we saw earlier, a shorter version of this would be `exact Eq.refl _`. But this situation comes up often enough that there is an even shorter version: the tactic `rfl` can be used as a shorthand for either `exact Eq.refl _` or `exact Iff.refl _`. In other words, in a tactic-mode proof, if the goal has one of the forms $a = a'$ or $a \leftrightarrow a'$, where a and a' are definitionally equal, then the tactic `rfl` will prove the goal. So `rfl` will finish off the justification of the second equation, and we can move on to the third.

Lean File

```
have h6 : 6 * (j - k) = n :=  
calc 6 * (j - k)  
_ = 6 * j - 6 * k := Int.mul_sub 6 j k  
_ = 3 * (2 * j) - 2 * (3 * k) := by  
    rewrite [←mul_assoc 3 2 j]  
    rewrite [←mul_assoc 2 3 k]  
    rfl  
    done  
_ = 3 * n - 2 * n := by  
  
    done  
_ = (3 - 2) * n := sorry  
_ = n := sorry
```

Tactic State in Infoview

n : \mathbb{Z}
h1 : $2 \mid n \wedge 3 \mid n$
h2 : $\exists (c : \mathbb{Z}),$
$n = 2 * c$
h3 : $\exists (c : \mathbb{Z}),$
$n = 3 * c$
j : \mathbb{Z}
h4 : $n = 2 * j$
k : \mathbb{Z}
h5 : $n = 3 * k$
$\vdash 3 * (2 * j) -$
$2 * (3 * k) =$
$3 * n - 2 * n$

To justify the third equation we have to substitute n for both $2 * j$ and $3 * k$. We can use `h4` and `h5` in the rewrite tactic to do this. In fact, we can do it in one step: you can put a list of proofs of equations or biconditionals inside the brackets, and the `rewrite` tactic will perform all of the replacements, one after another. In our case, the tactic `rewrite [←h4, ←h5]` will first replace $2 * j$ in the goal with n , and then it will replace $3 * k$ with n .

Lean File

```

have h6 : 6 * (j - k) = n :=
calc 6 * (j - k)
  - = 6 * j - 6 * k := Int.mul_sub 6 j k
  - = 3 * (2 * j) - 2 * (3 * k) := by
    rewrite [←mul_assoc 3 2 j]
    rewrite [←mul_assoc 2 3 k]
    rfl
    done
  - = 3 * n - 2 * n := by
    rewrite [←h4, ←h5]
    done
  - = (3 - 2) * n := sorry
  - = n := sorry

```

Tactic State in Infoview

```

n : ℤ
h1 : 2 | n ∧ 3 | n
h2 : ∃ (c : ℤ),
  n = 2 * c
h3 : ∃ (c : ℤ),
  n = 3 * c
j : ℤ
h4 : n = 2 * j
k : ℤ
h5 : n = 3 * k
⊢ 3 * n - 2 * n =
  3 * n - 2 * n

```

Of course, the `rfl` tactic will now finish off the justification of the third equation.

The fourth equation is $3 * n - 2 * n = (3 - 2) * n$. It looks like the algebraic law we need to justify this is a lot like the one that was used in the first equation, but with the multiplication to the right of the subtraction rather than to the left. It shouldn't be surprising, therefore, that the name of the theorem we need is `Int.sub_mul`. The command `#check @Int.sub_mul` gives the response

```
Int.sub_mul : ∀ (a b c : ℤ), (a - b) * c = a * c - b * c
```

so `Int.sub_mul 3 2 n` is a proof of $(3 - 2) * n = 3 * n - 2 * n$. But the fourth equation has the sides of this equation reversed, so to justify it we need `(Int.sub_mul 3 2 n).symm`.

Finally, the fifth equation is $(3 - 2) * n = n$. Why is this true? Because it is definitionally equal to $1 * n = n$. Is there a theorem to justify this last equation? One way to find the answer is to type in this example:

```
example (n : Int) : 1 * n = n := by apply?
```

Lean responds with `exact Int.one_mul n`, and `#check @Int.one_mul` yields

```
Int.one_mul : ∀ (a : ℤ), 1 * a = a
```

So `Int.one_mul n` should justify the last equation. Here's the complete calculational proof, where we have shortened the second step a bit by doing both `rewrites` in one step. When a tactic proof is short enough that it can be written on one line, we generally leave off `done`.

```

have h6 : 6 * (j - k) = n :=
calc 6 * (j - k)
_ = 6 * j - 6 * k := Int.mul_sub 6 j k
_ = 3 * (2 * j) - 2 * (3 * k) := by
    rewrite [←mul_assoc 3 2 j, ←mul_assoc 2 3 k]; rfl
_ = 3 * n - 2 * n := by rewrite [←h4, ←h5]; rfl
_ = (3 - 2) * n := (Int.sub_mul 3 2 n).symm
_ = n := Int.one_mul n

```

Whew! This example illustrates why algebraic reasoning in Lean can be difficult. But one reason why this proof was challenging is that we justified all of our steps from basic algebraic principles. Fortunately, there are more powerful tactics that can automate some algebraic reasoning. For example, the tactic `ring` can combine algebraic laws involving addition, subtraction, multiplication, and exponentiation with natural number exponents to prove many equations in one step. Also, the tactic `rw` is a variant of `rewrite` that automatically applies `rfl` after the rewriting if it can be used to finish the proof. Here's a shortened version of our calculational proof that uses these tactics.

```

have h6 : 6 * (j - k) = n :=
calc 6 * (j - k)
_ = 3 * (2 * j) - 2 * (3 * k) := by ring
_ = 3 * n - 2 * n := by rw [←h4, ←h5]
_ = n := by ring

```

By the way, the theorems `Int.mul_sub`, `Int.sub_mul`, and `Int.one_mul` that we used earlier are the integer versions of more general theorems `mul_sub`, `sub_mul`, and `one_mul`. The `#check` command tells us what these general theorems say:

```

@mul_sub : ∀ {α : Type u_1} [inst : NonUnitalNonAssocRing α]
            (a b c : α), a * (b - c) = a * b - a * c

@sub_mul : ∀ {α : Type u_1} [inst : NonUnitalNonAssocRing α]
            (a b c : α), (a - b) * c = a * c - b * c

@one_mul : ∀ {M : Type u_1} [inst : MulOneClass M]
            (a : M), 1 * a = a

```

The implicit arguments say that these theorems apply in any number system with the appropriate algebraic properties. We'll use the third theorem in our next example, which involves algebraic reasoning about real numbers. You can use the `#check` command to find the meanings of the other theorems we use in this proof.

```
theorem Example_3_5_4 (x : Real) (h1 : x ≤ x ^ 2) : x ≤ 0 ∨ 1 ≤ x := by
or_right with h2      --h2 : ¬x ≤ 0; Goal : 1 ≤ x
have h3 : 0 < x := lt_of_not_le h2
have h4 : 1 * x ≤ x * x :=
calc 1 * x
_ = x := one_mul x
_ ≤ x ^ 2 := h1
_ = x * x := by ring
show 1 ≤ x from le_of_mul_le_mul_right h4 h3
done
```

Exercises

1. theorem Exercise_3_3_18a (a b c : Int)
$$(h1 : a \mid b) (h2 : a \mid c) : a \mid (b + c) := \text{sorry}$$
2. Complete the following proof by justifying the steps in the calculational proof. Remember that you can use the tactic `demorgan` : ... to apply one of De Morgan's laws to just a part of the goal. You may also find the theorem `and_or_left` useful. (Use `#check` to see what the theorem says.)

```
theorem Exercise_3_4_6 (U : Type) (A B C : Set U) :
A \ (B ∩ C) = (A \ B) ∪ (A \ C) := by
apply Set.ext
fix x : U
show x ∈ A \ (B ∩ C) ↔ x ∈ A \ B ∪ A \ C from
calc x ∈ A \ (B ∩ C)
_ ↔ x ∈ A ∧ ¬(x ∈ B ∧ x ∈ C) := sorry
_ ↔ x ∈ A ∧ (x ∉ B ∨ x ∉ C) := sorry
_ ↔ (x ∈ A ∧ x ∉ B) ∨ (x ∈ A ∧ x ∉ C) := sorry
_ ↔ x ∈ (A \ B) ∪ (A \ C) := sorry
done
```

For the next exercise you will need the following definitions:

```
def even (n : Int) : Prop := ∃ (k : Int), n = 2 * k

def odd (n : Int) : Prop := ∃ (k : Int), n = 2 * k + 1
```

These definitions tell Lean that if `n` has type `Int`, then `even n` means $\exists (k : \text{Int}), n = 2 * k$ and `odd n` means $\exists (k : \text{Int}), n = 2 * k + 1$.

3. `theorem Exercise_3_4_10 (x y : Int)
 (h1 : odd x) (h2 : odd y) : even (x - y) := sorry`
4. `theorem Exercise_3_4_27a :
 ∀ (n : Int), 15 | n ↔ 3 | n ∧ 5 | n := sorry`
5. `theorem Like_Exercise_3_7_5 (U : Type) (F : Set (Set U))
 (h1 : ℙ (U₀ F) ⊆ U₀ { ℙ A | A ∈ F }) :
 ∃ (A : Set U), A ∈ F ∧ ∀ (B : Set U), B ∈ F → B ⊆ A := sorry`

4 Relations

4.1. Ordered Pairs and Cartesian Products

Section 4.1 of *How To Prove It* defines the *Cartesian product* $A \times B$ of two sets A and B to be the set of all ordered pairs (a, b) , where $a \in A$ and $b \in B$. However, in Lean, Cartesian product is an operation on *types*, not sets. If A and B are types, then $\mathsf{A} \times \mathsf{B}$ is the type of ordered pairs (a, b) , where a has type A and b has type B . (To enter the symbol \times in Lean, type `\times` or `\times`.) In other words, if you have $\mathsf{a} : \mathsf{A}$ and $\mathsf{b} : \mathsf{B}$, then (a, b) is an object of type $\mathsf{A} \times \mathsf{B}$. There is also notation for the first and second coordinates of an ordered pair. If p has type $\mathsf{A} \times \mathsf{B}$, then $\mathsf{p}.fst$ is the first coordinate of p , and $\mathsf{p}.snd$ is the second coordinate. You can also use the notation $\mathsf{p}.1$ for the first coordinate of p and $\mathsf{p}.2$ for the second coordinate. This means that $\mathsf{p} = (\mathsf{p}.fst, \mathsf{p}.snd) = (\mathsf{p}.1, \mathsf{p}.2)$.

4.2. Relations

Section 4.2 of *HTPI* defines a *relation from A to B* to be a subset of $A \times B$. In other words, if R is a relation from A to B , then R is a set whose elements are ordered pairs (a, b) , where $a \in A$ and $b \in B$. We will see in the next section that in Lean, it is convenient to use a somewhat different definition of relations. Nevertheless, we will take some time in this section to study sets of ordered pairs. If A and B are types, and R has type `Set (A × B)`, then R is a set whose elements are ordered pairs (a, b) , where a has type A and b has type B .

Section 4.2 of *HTPI* discusses several concepts concerning relations. Here is how these concepts are defined in *HTPI* (*HTPI* p. 183):

Definition 4.2.3. Suppose R is a relation from A to B . Then the *domain* of R is the set

$$\text{Dom}(R) = \{a \in A \mid \exists b \in B((a, b) \in R)\}.$$

The *range* of R is the set

$$\text{Ran}(R) = \{b \in B \mid \exists a \in A((a, b) \in R)\}.$$

The *inverse* of R is the relation R^{-1} from B to A defined as follows:

$$R^{-1} = \{(b, a) \in B \times A \mid (a, b) \in R\}.$$

4.2. Relations

Finally, suppose R is a relation from A to B and S is a relation from B to C . Then the *composition* of S and R is the relation $S \circ R$ from A to C defined as follows:

$$S \circ R = \{(a, c) \in A \times C \mid \exists b \in B((a, b) \in R \text{ and } (b, c) \in S)\}.$$

There are several examples in *HTPI* that illustrate these definitions. We will focus here on seeing how to work with these concepts in Lean.

We can write corresponding definitions in Lean as follows:

```
def Dom {A B : Type} (R : Set (A × B)) : Set A :=
{a : A | ∃ (b : B), (a, b) ∈ R}

def Ran {A B : Type} (R : Set (A × B)) : Set B :=
{b : B | ∃ (a : A), (a, b) ∈ R}

def inv {A B : Type} (R : Set (A × B)) : Set (B × A) :=
{(b, a) : B × A | (a, b) ∈ R}

def comp {A B C : Type}
(S : Set (B × C)) (R : Set (A × B)) : Set (A × C) :=
{(a, c) : A × C | ∃ (x : B), (a, x) ∈ R ∧ (x, c) ∈ S}
```

Definitions in Lean are introduced with the keyword `def`. In the definition of `Dom`, we have declared that `A` and `B` are implicit arguments and `R` is an explicit argument. That means that, in a Lean file containing these definitions, if we have `R : Set (A × B)`, then we can just write `Dom R` for the domain of `R`, and Lean will figure out for itself what `A` and `B` are. After the list of arguments there is a colon and then the type of `Dom R`, which is `Set A`. This is followed by `:=` and then the definition of `Dom R`. The definition says that `Dom R` is the set of all objects `a` of type `A` such that there is some `b` of type `B` with $(a, b) \in R$. This is a direct translation, into Lean's type-theory language, of the first part of Definition 4.2.3. The other three definitions are similar; they define `Ran R` to be the range of `R`, `inv R` to be the inverse of `R`, and `comp S R` to be the composition of `S` and `R`.

Here is the main theorem about these concepts, as stated in *HTPI* (*HTPI* p. 187):

Theorem 4.2.5. *Suppose R is a relation from A to B , S is a relation from B to C , and T is a relation from C to D . Then:*

1. $(R^{-1})^{-1} = R$.
2. $\text{Dom}(R^{-1}) = \text{Ran}(R)$.
3. $\text{Ran}(R^{-1}) = \text{Dom}(R)$.
4. $T \circ (S \circ R) = (T \circ S) \circ R$.

$$5. (S \circ R)^{-1} = R^{-1} \circ S^{-1}.$$

All five parts of this theorem follow directly from the definitions of the relevant concepts. In fact, in the first three parts, Lean recognizes the two sides of the equation as being definitionally equal, and therefore the tactic `rfl` proves those parts:

```
theorem Theorem_4_2_5_1 {A B : Type}
  (R : Set (A × B)) : inv (inv R) = R := by rfl

theorem Theorem_4_2_5_2 {A B : Type}
  (R : Set (A × B)) : Dom (inv R) = Ran R := by rfl

theorem Theorem_4_2_5_3 {A B : Type}
  (R : Set (A × B)) : Ran (inv R) = Dom R := by rfl
```

The fourth part will take a little more work to prove. We start the proof like this:

```
theorem Theorem_4_2_5_4 {A B C D : Type}
  (R : Set (A × B)) (S : Set (B × C)) (T : Set (C × D)) :
  comp T (comp S R) = comp (comp T S) R := by
  apply Set.ext
  fix (a, d) : A × D
  done
```

After the `apply Set.ext` tactic, the goal is

$$\forall (x : A \times D), x \in \text{comp } T (\text{comp } S R) \leftrightarrow x \in \text{comp } (\text{comp } T S) R$$

The next step should be to introduce an arbitrary object of type $A \times D$. We could just call this object x , but Lean lets us use a shortcut here. An object of type $A \times D$ must have the form of an ordered pair, where the first coordinate has type A and the second has type D . So Lean lets us write it as an ordered pair right away. That's what we've done in the second step, `fix (a, d) : A × D`. This tactic introduces two new variables into the proof, $a : A$ and $d : D$. (The proof in *HTPI* uses a similar shortcut. And we used a similar shortcut in the definitions of `inv R` and `comp R`, where the elements of these sets were written as ordered pairs.)

Here is the complete proof.

```
theorem Theorem_4_2_5_4 {A B C D : Type}
  (R : Set (A × B)) (S : Set (B × C)) (T : Set (C × D)) :
  comp T (comp S R) = comp (comp T S) R := by
  apply Set.ext
```

```

fix (a, d) : A × D
apply Iff.intro
· -- (→)
  assume h1 : (a, d) ∈ comp T (comp S R)
    --Goal : (a, d) ∈ comp (comp T S) R
  define      --Goal : ∃ (x : B), (a, x) ∈ R ∧ (x, d) ∈ comp T S
  define at h1 --h1 : ∃ (x : C), (a, x) ∈ comp S R ∧ (x, d) ∈ T
  obtain (c : C) (h2 : (a, c) ∈ comp S R ∧ (c, d) ∈ T) from h1
  have h3 : (a, c) ∈ comp S R := h2.left
  define at h3 --h3 : ∃ (x : B), (a, x) ∈ R ∧ (x, c) ∈ S
  obtain (b : B) (h4 : (a, b) ∈ R ∧ (b, c) ∈ S) from h3
  apply Exists.intro b --Goal : (a, b) ∈ R ∧ (b, d) ∈ comp T S
  apply And.intro h4.left --Goal : (b, d) ∈ comp T S
  define      --Goal : ∃ (x : C), (b, x) ∈ S ∧ (x, d) ∈ T
  show ∃ (x : C), (b, x) ∈ S ∧ (x, d) ∈ T from
    Exists.intro c (And.intro h4.right h2.right)
  done
· -- (←)
  assume h1 : (a, d) ∈ comp (comp T S) R
  define; define at h1
  obtain (b : B) (h2 : (a, b) ∈ R ∧ (b, d) ∈ comp T S) from h1
  have h3 : (b, d) ∈ comp T S := h2.right
  define at h3
  obtain (c : C) (h4 : (b, c) ∈ S ∧ (c, d) ∈ T) from h3
  apply Exists.intro c
  apply And.intro _ h4.right
  define
  show ∃ (x : B), (a, x) ∈ R ∧ (x, c) ∈ S from
    Exists.intro b (And.intro h2.left h4.left)
  done
done

```

Of course, if you have trouble reading this proof, you can enter it into Lean and see how the tactic state changes over the course of the proof.

Here is a natural way to start the proof of part 5:

```

theorem Theorem_4_2_5_5 {A B C : Type}
  (R : Set (A × B)) (S : Set (B × C)) :
  inv (comp S R) = comp (inv R) (inv S) := by
  apply Set.ext
  fix (c, a) : C × A

```

```

apply Iff.intro
· -- (→)
  assume h1 : (c, a) ∈ inv (comp S R)
    --Goal : (c, a) ∈ comp (inv R) (inv S)
  define at h1      --h1 : ∃ (x : B), (a, x) ∈ R ∧ (x, c) ∈ S
  define          --Goal : ∃ (x : B), (c, x) ∈ inv S ∧ (x, a) ∈ inv R
  obtain (b : B) (h2 : (a, b) ∈ R ∧ (b, c) ∈ S) from h1
  apply Exists.intro b           --Goal : (c, b) ∈ inv S ∧ (b, a) ∈ inv R
  done
· -- (←)

done
done

```

After the tactics `apply Set.ext` and `fix (c, a) : C × A`, the goal is $(c, a) \in \text{inv}(\text{comp } S R) \leftrightarrow (c, a) \in \text{comp}(\text{inv } R)(\text{inv } S)$. For the proof of the left-to-right direction, we assume $h1 : (c, a) \in \text{inv}(\text{comp } S R)$, and we must prove $(c, a) \in \text{comp}(\text{inv } R)(\text{inv } S)$. The definition of $h1$ is an existential statement, so we apply existential instantiation to obtain $b : B$ and $h2 : (a, b) \in R \wedge (b, c) \in S$. The definition of the goal is also an existential statement, and after the tactic `apply Exists.intro b`, the goal is $(c, b) \in \text{inv } S \wedge (b, a) \in \text{inv } R$. It looks like this goal will follow easily from $h2$, using the definitions of the inverses of S and R .

One way to write out these definitions would be to use the tactics `define : (c, b) ∈ inv S` and `define : (b, a) ∈ inv R`. But we're going to use this example to illustrate another way to proceed. To use this alternative method, we'll need to prove a preliminary theorem before proving part 5 of Theorem 4.2.5:

```

theorem inv_def {A B : Type} (R : Set (A × B)) (a : A) (b : B) :
  (b, a) ∈ inv R ↔ (a, b) ∈ R := by rfl

```

Now, any time we have a relation $R : \text{Set}(A \times B)$ and objects $a : A$ and $b : B$, the expression `inv_def R a b` will be a proof of the statement $(b, a) \in \text{inv } R \leftrightarrow (a, b) \in R$. (Note that A and B are implicit arguments and don't need to be specified.) And that means that the tactic `rewrite [inv_def R a b]` will change $(b, a) \in \text{inv } R$ to $(a, b) \in R$. In fact, as we've seen before, you can just write `rewrite [inv_def]`, and Lean will figure out how to apply the theorem `inv_def` to rewrite some part of the goal.

Returning to our proof of part 5 of Theorem 4.2.5, recall that after the step `apply Exists.intro b`, the goal is $(c, b) \in \text{inv } S \wedge (b, a) \in \text{inv } R$. Rather than using the `define` tactic to write out the definitions of the inverses, we'll use the tactic `rewrite [inv_def, inv_def]`. Why do we list `inv_def` twice in the `rewrite` tactic? When we ask Lean to use the theorem `inv_def` as a rewriting rule, it figures out that `inv_def S b c` is a proof of the statement $(c, b) \in \text{inv } S \leftrightarrow (b, c) \in S$, which can be used to rewrite the left half of the goal. To rewrite the right half,

we need a different application of the `inv_def` theorem, `inv_def R a b`. So we have to ask Lean to apply the theorem a second time. After the `rewrite` tactic, the goal is $(b, c) \in S \wedge (a, b) \in R$, which will follow easily from `h2`.

The rest of the proof is straightforward. Here is the complete proof.

```
theorem Theorem_4_2_5_5 {A B C : Type}
  (R : Set (A × B)) (S : Set (B × C)) :
  inv (comp S R) = comp (inv R) (inv S) := by
  apply Set.ext
  fix (c, a) : C × A
  apply Iff.intro
  · -- (→)
    assume h1 : (c, a) ∈ inv (comp S R)
      --Goal : (c, a) ∈ comp (inv R) (inv S)
    define at h1      --h1 : ∃ (x : B), (a, x) ∈ R ∧ (x, c) ∈ S
    define          --Goal : ∃ (x : B), (c, x) ∈ inv S ∧ (x, a) ∈ inv R
    obtain (b : B) (h2 : (a, b) ∈ R ∧ (b, c) ∈ S) from h1
    apply Exists.intro b      --Goal : (c, b) ∈ inv S ∧ (b, a) ∈ inv R
    rewrite [inv_def, inv_def] --Goal : (b, c) ∈ S ∧ (a, b) ∈ R
    show (b, c) ∈ S ∧ (a, b) ∈ R from And.intro h2.right h2.left
    done
  · -- (←)
    assume h1 : (c, a) ∈ comp (inv R) (inv S)
    define at h1
    define
    obtain (b : B) (h2 : (c, b) ∈ inv S ∧ (b, a) ∈ inv R) from h1
    apply Exists.intro b
    rewrite [inv_def, inv_def] at h2
    show (a, b) ∈ R ∧ (b, c) ∈ S from And.intro h2.right h2.left
    done
  done
```

By the way, an alternative way to complete both directions of this proof would have been to apply the commutativity of “and”. See if you can guess the name of that theorem (you can use `#check` to confirm your guess) and apply it as a third rewriting rule in the `rewrite` steps.

Exercises

1. `theorem Exercise_4_2_9a {A B C : Type} (R : Set (A × B)) (S : Set (B × C)) : Dom (comp S R) ⊆ Dom R := sorry`

2.

```
theorem Exercise_4_2_9b {A B C : Type} (R : Set (A × B))
  (S : Set (B × C)) : Ran R ⊆ Dom S → Dom (comp S R) = Dom R := sorry
```
3.

```
--Fill in the blank to get a correct theorem and then prove the theorem
theorem Exercise_4_2_9c {A B C : Type} (R : Set (A × B))
  (S : Set (B × C)) : ___ → Ran (comp S R) = Ran S := sorry
```
4.

```
theorem Exercise_4_2_12a {A B C : Type}
  (R : Set (A × B)) (S T : Set (B × C)) :
  (comp S R) \ (comp T R) ⊆ comp (S \ T) R := sorry
```

5. Here is an incorrect theorem with an incorrect proof.

Incorrect Theorem. Suppose R is a relation from A to B and S and T are relations from B to C . Then $(S \setminus T) \circ R \subseteq (S \circ R) \setminus (T \circ R)$.

Incorrect Proof (HTPI p. 190). Suppose $(a, c) \in (S \setminus T) \circ R$. Then we can choose some $b \in B$ such that $(a, b) \in R$ and $(b, c) \in S \setminus T$, so $(b, c) \in S$ and $(b, c) \notin T$. Since $(a, b) \in R$ and $(b, c) \in S$, $(a, c) \in S \circ R$. Similarly, since $(a, b) \in R$ and $(b, c) \notin T$, $(a, c) \notin T \circ R$. Therefore $(a, c) \in (S \circ R) \setminus (T \circ R)$. Since (a, c) was arbitrary, this shows that $(S \setminus T) \circ R \subseteq (S \circ R) \setminus (T \circ R)$. \square

Find the mistake in the proof by attempting to write the proof in Lean:

```
--You won't be able to complete this proof
theorem Exercise_4_2_12b {A B C : Type}
  (R : Set (A × B)) (S T : Set (B × C)) :
  comp (S \ T) R ⊆ (comp S R) \ (comp T R) := sorry
```

6. Is the following theorem correct? Try to prove it in Lean. If you can't prove it, see if you can find a counterexample.

```
--You might not be able to complete this proof
theorem Exercise_4_2_14c {A B C : Type}
  (R : Set (A × B)) (S T : Set (B × C)) :
  comp (S ∩ T) R = (comp S R) ∩ (comp T R) := sorry
```

7. Is the following theorem correct? Try to prove it in Lean. If you can't prove it, see if you can find a counterexample.

```
--You might not be able to complete this proof
theorem Exercise_4_2_14d {A B C : Type}
  (R : Set (A × B)) (S T : Set (B × C)) :
  comp (S ∪ T) R = (comp S R) ∪ (comp T R) := sorry
```

4.3. More About Relations

Section 4.3 of *HTPI* introduces new notation for working with relations. If $R \subseteq A \times B$, $a \in A$, and $b \in B$, then *HTPI* introduces the notation aRb as an alternative way of saying $(a, b) \in R$.

The notation we will use in Lean is slightly different. Corresponding to the notation aRb in *HTPI*, in Lean we will use the notation $R a b$. And we cannot use this notation when R has type $\text{Set } (A \times B)$. Rather, we will need to introduce a new type for the variable R in the notation $R a b$. The name we will use for this new type is $\text{Rel } A B$. Thus, if R has type $\text{Rel } A B$, a has type A , and b has type B , then $R a b$ is a proposition. This should remind you of the way predicates work in Lean. If we have $P : \text{Pred } A$, then we think of P as representing a property that an object of type A might have, and if we also have $a : A$, then $P a$ is the proposition asserting that a has the property represented by P . Similarly, if we have $R : \text{Rel } A B$, then we can think of R as representing a relationship that might hold between an object of type A and an object of type B , and if we also have $a : A$ and $b : B$, then $R a b$ is the proposition asserting that the relationship represented by R holds between a and b .

Notice that in *HTPI*, the same variable R is used in both the notation aRb and $(a, b) \in R$. But in Lean, the notation $R a b$ is used when R has type $\text{Rel } A B$, and the notation $(a, b) \in R$ is used when R has type $\text{Set } (A \times B)$. The types $\text{Rel } A B$ and $\text{Set } (A \times B)$ are different, so we cannot use the same variable R in the two notations. However, there is a correspondence between the two types. Suppose R has type $\text{Rel } A B$. If we let R' denote the set of all ordered pairs $(a, b) : A \times B$ such that the proposition $R a b$ is true, then R' has type $\text{Set } (A \times B)$. And there is then a simple relationship between R and R' : for any objects $a : A$ and $b : B$, the propositions $R a b$ and $(a, b) \in R'$ are equivalent. For our work in Lean, we will say that R is a *relation* from A to B , and R' is the *extension* of R .

We can define the extension of a relation, and state the correspondence between a relation and its extension, in Lean as follows:

```
def extension {A B : Type} (R : Rel A B) : Set (A × B) :=
  {(a, b) : A × B | R a b}

theorem ext_def {A B : Type} (R : Rel A B) (a : A) (b : B) :
  (a, b) ∈ extension R ↔ R a b := by rfl
```

4.3. More About Relations

The rest of Chapter 4 of *HTPI* focuses on relations from a set to itself; in Lean, the corresponding idea is a relation from a type to itself. If A is any type and R has type $\text{Rel } A \ A$, then we will say that R is a *binary relation on A* . The notation $\text{BinRel } A$ denotes the type of binary relations on A . In other words, $\text{BinRel } A$ is just an abbreviation for $\text{Rel } A \ A$. If R is a binary relation on A , then we say that R is *reflexive* if for every x of type A , $R \ x \ x$ holds. It is *symmetric* if for all x and y of type A , if $R \ x \ y$ then $R \ y \ x$. And it is *transitive* if for all x , y , and z of type A , if $R \ x \ y$ and $R \ y \ z$ then $R \ x \ z$. Of course, we can tell Lean about these definitions, which correspond to Definition 4.3.2 in *HTPI*:

```
def reflexive {A : Type} (R : BinRel A) : Prop :=
  ∀ (x : A), R x x

def symmetric {A : Type} (R : BinRel A) : Prop :=
  ∀ (x y : A), R x y → R y x

def transitive {A : Type} (R : BinRel A) : Prop :=
  ∀ (x y z : A), R x y → R y z → R x z
```

Once again, we refer you to *HTPI* to see examples of these concepts, and we focus here on proving theorems about these concepts in Lean. The main theorem about these concepts in Section 4.3 of *HTPI* is Theorem 4.3.4. Here is what it says (*HTPI* p. 196):

Theorem 4.3.4. *Suppose R is a relation on a set A .*

1. *R is reflexive iff $\{(x, y) \in A \times A \mid x = y\} \subseteq R$.*
2. *R is symmetric iff $R = R^{-1}$.*
3. *R is transitive iff $R \circ R \subseteq R$.*

We can prove corresponding statements in Lean, but we'll have to be careful to distinguish between the types $\text{BinRel } A$ and $\text{Set } (A \times A)$. In *HTPI*, each of the three statements in the theorem uses the same letter R on both sides of the “iff”, but we can't write the statements that way in Lean. In each statement, the part before “iff” uses a concept that was defined for objects of type $\text{BinRel } A$, whereas the part after “iff” uses concepts that only make sense for objects of type $\text{Set } (A \times A)$. So we'll have to rephrase the statements by using the correspondence between a relation of type $\text{BinRel } A$ and its extension, which has type $\text{Set } (A \times A)$. Here's the Lean theorem corresponding to statement 2 of Theorem 4.3.4:

```
theorem Theorem_4_3_4_2 {A : Type} (R : BinRel A) :
  symmetric R ↔ extension R = inv (extension R) := by
  apply Iff.intro
  · -- (→)
    assume h1 : symmetric R
```

```

define at h1           --h1 : ∀ (x y : A), R x y → R y x
apply Set.ext
fix (a, b) : A × A
show (a, b) ∈ extension R ↔ (a, b) ∈ inv (extension R) from
calc (a, b) ∈ extension R
  _ ↔ R a b := by rfl
  _ ↔ R b a := Iff.intro (h1 a b) (h1 b a)
  _ ↔ (a, b) ∈ inv (extension R) := by rfl
done
· -- (←)
assume h1 : extension R = inv (extension R)
define                      --Goal : ∀ (x y : A), R x y → R y x
fix a : A; fix b : A
assume h2 : R a b          --Goal : R b a
rewrite [←ext_def R, h1, inv_def, ext_def] at h2
show R b a from h2
done
done

```

Note that near the end of the proof, we assume $h2 : R a b$, and our goal is $R b a$. We convert $R a b$ to $R b a$ by a sequence of rewrites. Applying the right-to-left direction of the theorem $\text{ext_def } R a b$ converts $R a b$ to $(a, b) \in \text{extension } R$. Then rewriting with $h1$ converts this to $(a, b) \in \text{inv } (\text{extension } R)$, using $\text{inv_def } (\text{extension } R) b a$ converts this to $(b, a) \in \text{extension } R$, and finally $\text{ext_def } R b a$ produces $R b a$. Usually we can leave out the arguments when we use a theorem as a rewriting rule, and Lean will figure them out for itself. But in this case, if you try using ext_def as the first rewriting rule, you will see that Lean is unable to figure out that it should use the right-to-left direction of $\text{ext_def } R a b$. Supplying the first argument turns out to be enough of a hint for Lean to figure out the rest. That's why our first rewriting rule is $\text{ext_def } R$.

We'll leave the proofs of the other two statements in Theorem 4.3.4 as exercises for you.

For any types A and B , if we want to define a particular relation R from A to B , we can do it by specifying, for any $a : A$ and $b : B$, what proposition is represented by $R a b$. For example, for any type A , we can define a relation `elementhood A` from A to `Set A` as follows:

```
def elementhood (A : Type) (a : A) (X : Set A) : Prop := a ∈ X
```

This definition says that if A is a type, a has type A , and X has type `Set A`, then `elementhood A a X` is the proposition $a \in X$. Thus, if `elementhood A` is followed by objects of type A and `Set A`, the result is a proposition, so `elementhood A` is functioning as a relation from A to `Set A`. For example, `elementhood Int` is a relation from integers to sets of integers, and `elementhood Int`

$6 \{n : \text{Int} \mid \exists (k : \text{Int}), n = 2 * k\}$ is the (true) statement that 6 is an element of the set of even integers. (You are asked to prove it in the exercises.)

We can also use this method to define an operation that reverses the process of forming the extension of a relation. If R has type $\text{Set } (A \times B)$, then we define $\text{RelFromExt } R$ to be the relation whose extension is R . A few simple theorems, which follow directly from the definition, clarify the meaning of $\text{RelFromExt } R$.

```
def RelFromExt {A B : Type}
  (R : Set (A × B)) (a : A) (b : B) : Prop := (a, b) ∈ R

theorem RelFromExt_def {A B : Type}
  (R : Set (A × B)) (a : A) (b : B) :
  RelFromExt R a b ↔ (a, b) ∈ R := by rfl

example {A B : Type} (R : Rel A B) :
  RelFromExt (extension R) = R := by rfl

example {A B : Type} (R : Set (A × B)) :
  extension (RelFromExt R) = R := by rfl
```

Exercises

1. `example :`
elementhood Int 6 {n : Int | ∃ (k : Int), n = 2 * k} := sorry
2. `theorem Theorem_4_3_4_1 {A : Type} (R : BinRel A) :`
reflexive R ↔ {(x, y) : A × A | x = y} ⊆ extension R := sorry
3. `theorem Theorem_4_3_4_3 {A : Type} (R : BinRel A) :`
transitive R ↔ comp (extension R) (extension R) ⊆ extension R := sorry
4. `theorem Exercise_4_3_12a {A : Type} (R : BinRel A) (h1 : reflexive R) :`
reflexive (RelFromExt (inv (extension R))) := sorry
5. `theorem Exercise_4_3_12c {A : Type} (R : BinRel A) (h1 : transitive R) :`
transitive (RelFromExt (inv (extension R))) := sorry

6.

```
theorem Exercise_4_3_18 {A : Type}
  (R S : BinRel A) (h1 : transitive R) (h2 : transitive S)
  (h3 : comp (extension S) (extension R) ⊆
    comp (extension R) (extension S)) :
  transitive (RelFromExt (comp (extension R) (extension S))) := sorry
```
7.

```
theorem Exercise_4_3_20 {A : Type} (R : BinRel A) (S : BinRel (Set A))
  (h : ∀ (X Y : Set A), S X Y ↔ X ≠ ∅ ∧ Y ≠ ∅ ∧
  ∀ (x y : A), x ∈ X → y ∈ Y → R x y) :
  transitive R → transitive S := sorry
```

In the next three exercises, determine whether or not the theorem is correct.

8.

```
--You might not be able to complete this proof
theorem Exercise_4_3_13b {A : Type}
  (R1 R2 : BinRel A) (h1 : symmetric R1) (h2 : symmetric R2) :
  symmetric (RelFromExt ((extension R1) ∪ (extension R2))) := sorry
```
9.

```
--You might not be able to complete this proof
theorem Exercise_4_3_13c {A : Type}
  (R1 R2 : BinRel A) (h1 : transitive R1) (h2 : transitive R2) :
  transitive (RelFromExt ((extension R1) ∪ (extension R2))) := sorry
```
10.

```
--You might not be able to complete this proof
theorem Exercise_4_3_19 {A : Type} (R : BinRel A) (S : BinRel (Set A))
  (h : ∀ (X Y : Set A), S X Y ↔ ∃ (x y : A), x ∈ X ∧ y ∈ Y ∧ R x y) :
  transitive R → transitive S := sorry
```

4.4. Ordering Relations

Section 4.4 of *HTPI* begins by defining several new concepts about binary relations. Here are the definitions, written in Lean:

```
def antisymmetric {A : Type} (R : BinRel A) : Prop :=
  ∀ (x y : A), R x y → R y x → x = y

def partial_order {A : Type} (R : BinRel A) : Prop :=
  reflexive R ∧ transitive R ∧ antisymmetric R

def total_order {A : Type} (R : BinRel A) : Prop :=
  partial_order R ∧ ∀ (x y : A), R x y ∨ R y x
```

4.4. Ordering Relations

These definitions say that if R is a binary relation on A , then R is *antisymmetric* if $R \ x \ y$ and $R \ y \ x$ cannot both be true unless $x = y$. R is a *partial order on A*—or just a *partial order*, if A is clear from context—if it is reflexive, transitive, and antisymmetric. And R is a *total order on A* if it is a partial order and also, for any x and y of type A , either $R \ x \ y$ or $R \ y \ x$. Note that, since Lean groups the connective \wedge to the right, `partial_order R` means `reflexive R ∧ (transitive R ∧ antisymmetric R)`, and therefore if `h` is a proof of `partial_order R`, then `h.left` is a proof of `reflexive R`, `h.right.left` is a proof of `transitive R`, and `h.right.right` is a proof of `antisymmetric R`.

Example 4.4.3 in *HTPI* gives several examples of partial orders and total orders. We'll give one of those examples here. For any type A , we define `sub A` to be the subset relation on sets of objects of type A :

```
def sub (A : Type) (X Y : Set A) : Prop := X ⊆ Y
```

According to this definition, `sub A` is a binary relation on `Set A`, and for any two sets X and Y of type `Set A`, `sub A X Y` is the proposition $X \subseteq Y$. We will leave it as an exercise for you to prove that `sub A` is a partial order on the type `Set A`.

Notice that $X \subseteq Y$ could be thought of as expressing a sense in which Y is “at least as large as” X . Often, if R is a partial order on A and a and b have type A , then $R \ a \ b$ can be thought of as meaning that b is in some sense “at least as large as” a . Many of the concepts we study for partial and total orders are motivated by this interpretation of R .

For example, if R is a partial order on A , B has type `Set A`, and b has type A , then we say that b is an *R-smallest element* of B if it is an element of B , and every element of B is at least as large as b , according to this interpretation of the ordering R . We say that b is an *R-minimal element* of B if it is an element of B , and there is no other element of B that is smaller than b , according to the ordering R . We can state these precisely as definitions in Lean:

```
def smallestElt {A : Type} (R : BinRel A) (b : A) (B : Set A) : Prop :=
  b ∈ B ∧ ∀ x ∈ B, R b x

def minimalElt {A : Type} (R : BinRel A) (b : A) (B : Set A) : Prop :=
  b ∈ B ∧ ¬∃ x ∈ B, R x b ∧ x ≠ b
```

Notice that, as in *HTPI*, in Lean we can write $\forall x \in B, P x$ as an abbreviation for $\forall (x : A), x \in B \rightarrow P x$, and $\exists x \in B, P x$ as an abbreviation for $\exists (x : A), x \in B \wedge P x$. According to these definitions, if R is a partial order then `smallestElt R b B` is the proposition that b is an R -smallest element of B , and `minimalElt R b B` means that b is an R -minimal element of B . Although the definitions do not explicitly say that R must be a partial order, we will only use them in situations in which that is the case.

4.4. Ordering Relations

Theorem 4.4.6 in *HTPI* asserts three statements about these concepts. We'll prove the second and third, and leave the first as an exercise for you. The first statement in Theorem 4.4.6 says that if B has an R -smallest element, then that R -smallest element is unique. Thus, we can talk about *the* R -smallest element of B rather than *an* R -smallest element. The second says that if b is the R -smallest element of B , then it is also an R -minimal element, and it is the only R -minimal element. Here is how you might start the proof. (Although Lean sometimes uses bounded quantifiers as abbreviations in the Infoview, we have written out the unabbreviated statements in the comments, to make the logic of some steps easier to follow.)

```
theorem Theorem_4_4_6_2 {A : Type} (R : BinRel A) (B : Set A) (b : A)
  (h1 : partial_order R) (h2 : smallestElt R b B) :
  minimalElt R b B ∧ ∀ (c : A), minimalElt R c B → b = c := by
  define at h1      --h1 : reflexive R ∧ transitive R ∧ antisymmetric R
  define at h2      --h2 : b ∈ B ∧ ∀ (x : A), x ∈ B → R b x
  apply And.intro
  . -- Proof that b is minimal
    define          --Goal : b ∈ B ∧ ¬∃ (x : A), x ∈ B ∧ R x b ∧ x ≠ b
    apply And.intro h2.left
    quant_neg      --Goal : ∀ (x : A), ¬(x ∈ B ∧ R x b ∧ x ≠ b)
    demorgan : ¬(x ∈ B ∧ R x b ∧ x ≠ b)
    done
  . -- Proof that b is only minimal element
    done
done
```

When the goal is $\forall (x : A), \neg(x \in B \wedge R x b \wedge x \neq b)$, it is tempting to apply the `demorgan` tactic to $\neg(x \in B \wedge R x b \wedge x \neq b)$, but unfortunately this generates an error in Lean: unknown identifier 'x'. The problem is that x is not defined in the tactic state, so without the quantifier $\forall (x : A)$ in front of it, $\neg(x \in B \wedge R x b \wedge x \neq b)$ doesn't mean anything to Lean. The solution to the problem is to deal with the universal quantifier first by introducing an arbitrary x of type A . Once x has been introduced, we can apply the `demorgan` tactic.

```
theorem Theorem_4_4_6_2 {A : Type} (R : BinRel A) (B : Set A) (b : A)
  (h1 : partial_order R) (h2 : smallestElt R b B) :
  minimalElt R b B ∧ ∀ (c : A), minimalElt R c B → b = c := by
  define at h1      --h1 : reflexive R ∧ transitive R ∧ antisymmetric R
  define at h2      --h2 : b ∈ B ∧ ∀ (x : A), x ∈ B → R b x
  apply And.intro
  . -- Proof that b is minimal
    define          --Goal : b ∈ B ∧ ¬∃ (x : A), x ∈ B ∧ R x b ∧ x ≠ b
    apply And.intro h2.left
```

4.4. Ordering Relations

```

quant_neg      --Goal : ∀ (x : A), ¬(x ∈ B ∧ R x b ∧ x ≠ b)
fix x : A
demorgan      --Goal : ¬x ∈ B ∨ ¬(R x b ∧ x ≠ b)
or_right with h3 --h3 : x ∈ B; Goal : ¬(R x b ∧ x ≠ b)
demorgan      --Goal : ¬R x b ∨ x = b
or_right with h4 --h4 : R x b; Goal : x = b
have h5 : R b x := h2.right x h3
have h6 : antisymmetric R := h1.right.right
define at h6   --h6 : ∀ (x y : A), R x y → R y x → x = y
show x = b from h6 x b h4 h5
done
• -- Proof that b is only minimal element
fix c : A
assume h3 : minimalElt R c B
define at h3   --h3 : c ∈ B ∧ ¬∃ (x : A), x ∈ B ∧ R x c ∧ x ≠ c
contradict h3.right with h4
          --h4 : ¬b = c; Goal : ∃ (x : A), x ∈ B ∧ R x c ∧ x ≠ c
have h5 : R b c := h2.right c h3.left
show ∃ (x : A), x ∈ B ∧ R x c ∧ x ≠ c from
    Exists.intro b (And.intro h2.left (And.intro h5 h4))
done
done

```

Finally, the third statement in Theorem 4.4.6 says that if R is a total order, then any R -minimal element of a set B must be the R -smallest element of B . The beginning of the proof is straightforward:

```

theorem Theorem_4_4_6_3 {A : Type} (R : BinRel A) (B : Set A) (b : A)
  (h1 : total_order R) (h2 : minimalElt R b B) : smallestElt R b B := by
  define at h1      --h1 : partial_order R ∧ ∀ (x y : A), R x y ∨ R y x
  define at h2      --h2 : b ∈ B ∧ ¬∃ (x : A), x ∈ B ∧ R x b ∧ x ≠ b
  define          --Goal : b ∈ B ∧ ∀ (x : A), x ∈ B → R b x
  apply And.intro h2.left --Goal : ∀ (x : A), x ∈ B → R b x
  fix x : A
  assume h3 : x ∈ B      --Goal : R b x
  done

```

Surprisingly, at this point it is difficult to find a way to reach the goal $R b x$. See *HTPI* for an explanation of why it turns out to be helpful to split the proof into two cases, depending on whether or not $x = b$. Of course, we use the `by_cases` tactic for this.

```

theorem Theorem_4_4_6_3 {A : Type} (R : BinRel A) (B : Set A) (b : A)
  (h1 : total_order R) (h2 : minimalElt R b B) : smallestElt R b B := by
  define at h1      --h1 : partial_order R ∧ ∀ (x y : A), R x y ∨ R y x
  define at h2      --h2 : b ∈ B ∧ ¬∃ (x : A), x ∈ B ∧ R x b ∧ x ≠ b
  define           --Goal : b ∈ B ∧ ∀ (x : A), x ∈ B → R b x
  apply And.intro h2.left --Goal : ∀ (x : A), x ∈ B → R b x
  fix x : A
  assume h3 : x ∈ B      --Goal : R b x
  by_cases h4 : x = b
  · -- Case 1. h4 : x = b
    rewrite [h4]          --Goal : R b b
    have h5 : partial_order R := h1.left
    define at h5
    have h6 : reflexive R := h5.left
    define at h6
    show R b b from h6 b
    done
  · -- Case 2. h4 : x ≠ b
    have h5 : ∀ (x y : A), R x y ∨ R y x := h1.right
    have h6 : R x b ∨ R b x := h5 x b
    have h7 : ¬R x b := by
      contradict h2.right with h8
      show ∃ (x : A), x ∈ B ∧ R x b ∧ x ≠ b from
        Exists.intro x (And.intro h3 (And.intro h8 h4))
      done
    disj_syll h6 h7
    show R b x from h6
    done
  done

```

Imitating the definitions above, you should be able to formulate definitions of R -largest and R -maximal elements. Section 4.4 of *HTPI* defines four more terms: upper bound, lower bound, least upper bound, and greatest lower bound. We will discuss upper bounds and least upper bounds, and leave lower bounds and greatest lower bounds for you to figure out on your own.

If R is a partial order on A , B has type $\text{Set } A$, and a has type A , then a is called an *upper bound* for B if it is at least as large as every element of B . If it is the smallest element of the set of upper bounds, then it is called the *least upper bound* of B . The phrase “least upper bound” is often abbreviated “lub”. Here are these definitions, written in Lean:

```

def upperBd {A : Type} (R : BinRel A) (a : A) (B : Set A) : Prop :=
  ∀ x ∈ B, R x a

```

```
def lub {A : Type} (R : BinRel A) (a : A) (B : Set A) : Prop :=
  smallestElt R a {c : A | upperBd R c B}
```

As usual, we will let you consult *HTPI* for examples of these concepts. But we will mention one example: If A is a type and F has type $\text{Set}(\text{Set } A)$ —that is, F is a set whose elements are sets of objects of type A —then the least upper bound of F , with respect to the partial order $\text{sub } A$, is $\cup_0 F$. We leave the proof of this fact as an exercise.

Exercises

1. `theorem Example_4_4_3_1 {A : Type} : partial_order (sub A) := sorry`
2. `theorem Theorem_4_4_6_1 {A : Type} (R : BinRel A) (B : Set A) (b : A)
 (h1 : partial_order R) (h2 : smallestElt R b B) :
 ∀ (c : A), smallestElt R c B → b = c := sorry`
3. `--If F is a set of sets, then $\cup_0 F$ is the lub of F in the subset ordering
 theorem Theorem_4_4_11 {A : Type} (F : Set (Set A)) :
 lub (sub A) (F) F := sorry`
4. `theorem Exercise_4_4_8 {A B : Type} (R : BinRel A) (S : BinRel B)
 (T : BinRel (A × B)) (h1 : partial_order R) (h2 : partial_order S)
 (h3 : ∀ (a a' : A) (b b' : B),
 T (a, b) (a', b') ↔ R a a' ∧ S b b') :
 partial_order T := sorry`
5. `theorem Exercise_4_4_9_part {A B : Type} (R : BinRel A) (S : BinRel B)
 (L : BinRel (A × B)) (h1 : total_order R) (h2 : total_order S)
 (h3 : ∀ (a a' : A) (b b' : B),
 L (a, b) (a', b') ↔ R a a' ∧ (a = a' → S b b')) :
 ∀ (a a' : A) (b b' : B),
 L (a, b) (a', b') ∨ L (a', b') (a, b) := sorry`
6. `theorem Exercise_4_4_15a {A : Type}
 (R1 R2 : BinRel A) (B : Set A) (b : A)
 (h1 : partial_order R1) (h2 : partial_order R2)
 (h3 : extension R1 ⊆ extension R2) :
 smallestElt R1 b B → smallestElt R2 b B := sorry`

7. `theorem Exercise_4_4_15b {A : Type}`
 $(R1 R2 : \text{BinRel } A) (B : \text{Set } A) (b : A)$
 $(h1 : \text{partial_order } R1) (h2 : \text{partial_order } R2)$
 $(h3 : \text{extension } R1 \subseteq \text{extension } R2) :$
 $\text{minimalElt } R2 b B \rightarrow \text{minimalElt } R1 b B := \text{sorry}$
8. `theorem Exercise_4_4_18a {A : Type}`
 $(R : \text{BinRel } A) (B1 B2 : \text{Set } A) (h1 : \text{partial_order } R)$
 $(h2 : \forall x \in B1, \exists y \in B2, R x y) (h3 : \forall x \in B2, \exists y \in B1, R x y) :$
 $\forall (x : A), \text{upperBd } R x B1 \leftrightarrow \text{upperBd } R x B2 := \text{sorry}$
9. `theorem Exercise_4_4_22 {A : Type}`
 $(R : \text{BinRel } A) (B1 B2 : \text{Set } A) (x1 x2 : A)$
 $(h1 : \text{partial_order } R) (h2 : \text{lub } R x1 B1) (h3 : \text{lub } R x2 B2) :$
 $B1 \subseteq B2 \rightarrow R x1 x2 := \text{sorry}$
10. `theorem Exercise_4_4_24 {A : Type} (R : \text{Set } (A \times A)) :`
 $\text{smallestElt } (\text{sub } (A \times A)) (R \cup (\text{inv } R))$
 $\{T : \text{Set } (A \times A) \mid R \subseteq T \wedge \text{symmetric } (\text{RelFromExt } T)\} := \text{sorry}$

4.5. Equivalence Relations

Chapter 4 of *HTPI* concludes with the study of one more important combination of properties that a relation might have. A binary relation R on a set A is called an *equivalence relation* if it is reflexive, symmetric, and transitive. If $x \in A$, then the *equivalence class* of x with respect to R is the set of all $y \in A$ such that yRx . In *HTPI*, this equivalence class is denoted $[x]_R$, so we have

$$[x]_R = \{y \in A \mid yRx\}.$$

The set whose elements are all of these equivalence classes is called A mod R . It is written A/R , so

$$A/R = \{[x]_R \mid x \in A\}.$$

Note that A/R is a set whose elements are sets: for each $x \in A$, $[x]_R$ is a subset of A , and $[x]_R \in A/R$.

To define these concepts in Lean, we write:

```
def equiv_rel {A : Type} (R : BinRel A) : Prop :=
reflexive R ∧ symmetric R ∧ transitive R

def equivClass {A : Type} (R : BinRel A) (x : A) : Set A :=
{y : A | R y x}
```

```
def mod (A : Type) (R : BinRel A) : Set (Set A) :=
{equivClass R x | x : A}
```

Thus, `equiv_rel R` is the proposition that R is an equivalence relation, `equivClass R x` is the equivalence class of x with respect to R , and `mod A R` is A mod R . Note that `equivClass R x` has type `Set A`, while `mod A R` has type `Set (Set A)`. The definition of `mod A R` is shorthand for $\{X : Set A \mid \exists (x : A), \text{equivClass } R x = X\}$.

HTPI gives several examples of equivalence relations, and these examples illustrate that equivalence classes always have certain properties. The most important of these are that each equivalence class is a nonempty set, the equivalence classes do not overlap, and their union is all of A . We say that the equivalence classes form a *partition* of A . To state and prove these properties in Lean we will need some definitions. We start with these:

```
def empty {A : Type} (X : Set A) : Prop :=  $\neg\exists (x : A), x \in X$ 

def pairwise_disjoint {A : Type} (F : Set (Set A)) : Prop :=
 $\forall X \in F, \forall Y \in F, X \neq Y \rightarrow \text{empty } (X \cap Y)$ 
```

To say that a set X is empty, we could write $X = \emptyset$, but it is more convenient to have a statement that says more explicitly what it means for a set to be empty. Thus, we have defined `empty X` to be the proposition saying that X has no elements. If F has type `Set (Set A)`, then `pairwise_disjoint F` is the proposition that no two distinct elements of F have any element in common—in other words, the elements of F do not overlap. We can now give the precise definition of a partition:

```
def partition {A : Type} (F : Set (Set A)) : Prop :=
 $(\forall (x : A), x \in \bigcup F) \wedge \text{pairwise\_disjoint } F \wedge \forall X \in F, \neg\text{empty } X$ 
```

The main theorem about equivalence relations in *HTPI* is Theorem 4.5.4, which says that `mod A R` is a partition of A . The proof of this theorem is hard enough that *HTPI* proves two facts about equivalence classes first. A fact that is proven just for the purpose of using it to prove something else is often called a *lemma*. We can use this term in Lean as well. Here is the first part of Lemma 4.5.5 from *HTPI*

```
lemma Lemma_4_5_5_1 {A : Type} (R : BinRel A) (h : equiv_rel R) :
 $\forall (x : A), x \in \text{equivClass } R x := \text{by}$ 
fix x : A
define      --Goal : R x x
define at h    --h : reflexive R ∧ symmetric R ∧ transitive R
have Rref : reflexive R := h.left
```

```
show R x x from Rref x
done
```

The command `#check @Lemma_4_5_5_1` produces the result

```
@Lemma_4_5_5_1 : ∀ {A : Type} (R : BinRel A),
equiv_rel R → ∀ (x : A), x ∈ equivClass R x
```

Thus, if we have $R : \text{BinRel } A$, $h : \text{equiv_rel } R$, and $x : A$, then `Lemma_4_5_5_1 R h x` is a proof of $x \in \text{equivClass } R x$. We will use this at the end of the proof of our next lemma:

```
lemma Lemma_4_5_5_2 {A : Type} (R : BinRel A) (h : equiv_rel R) :
  ∀ (x y : A), y ∈ equivClass R x ↔
    equivClass R y = equivClass R x := by
  have Rsymm : symmetric R := h.right.left
  have Rtrans : transitive R := h.right.right
  fix x : A; fix y : A
  apply Iff.intro
  · -- (→)
    assume h2 :
      y ∈ equivClass R x      --Goal : equivClass R y = equivClass R x
    define at h2                  --h2 : R y x
    apply Set.ext
    fix z : A
    apply Iff.intro
    · -- Proof that  $z \in \text{equivClass } R y \rightarrow z \in \text{equivClass } R x$ 
      assume h3 : z ∈ equivClass R y
      define                                --Goal : R z x
      define at h3                      --h3 : R z y
      show R z x from Rtrans z y x h3 h2
      done
    · -- Proof that  $z \in \text{equivClass } R x \rightarrow z \in \text{equivClass } R y$ 
      assume h3 : z ∈ equivClass R x
      define                                --Goal : R z y
      define at h3                      --h3 : R z x
      have h4 : R x y := Rsymm y x h2
      show R z y from Rtrans z x y h3 h4
      done
    done
  · -- (←)
    assume h2 :
      equivClass R y = equivClass R x      --Goal : y ∈ equivClass R x
```

```

rewrite [←h2]                                     --Goal : y ∈ equivClass R y
show y ∈ equivClass R y from Lemma_4_5_5_1 R h y
done
done

```

The definition of “partition” has three parts, so to prove Theorem 4.5.4 we will have to prove three statements. It will make the proof easier to read if we prove the three statements separately.

```

lemma Theorem_4_5_4_part_1 {A : Type} (R : BinRel A) (h : equiv_rel R) :
  ∀ (x : A), x ∈ U₀ (mod A R) := by
  fix x : A
  define      --Goal : ∃ (t : Set A), t ∈ mod A R ∧ x ∈ t
  apply Exists.intro (equivClass R x)
  apply And.intro _ (Lemma_4_5_5_1 R h x)
    --Goal : equivClass R x ∈ mod A R
  define      --Goal : ∃ (x₁ : A), equivClass R x₁ = equivClass R x
  apply Exists.intro x
  rfl
  done

lemma Theorem_4_5_4_part_2 {A : Type} (R : BinRel A) (h : equiv_rel R) :
  pairwise_disjoint (mod A R) := by
  define
  fix X : Set A
  assume h2 : X ∈ mod A R
  fix Y : Set A
  assume h3 : Y ∈ mod A R           --Goal : X ≠ Y → empty (X ∩ Y)
  define at h2; define at h3
  obtain (x : A) (h4 : equivClass R x = X) from h2
  obtain (y : A) (h5 : equivClass R y = Y) from h3
  contrapos
  assume h6 : ∃ (x : A), x ∈ X ∩ Y --Goal : X = Y
  obtain (z : A) (h7 : z ∈ X ∩ Y) from h6
  define at h7
  rewrite [←h4, ←h5] at h7 --h7 : z ∈ equivClass R x ∧ z ∈ equivClass R y
  have h8 : equivClass R z = equivClass R x :=
    (Lemma_4_5_5_2 R h x z).ltr h7.left
  have h9 : equivClass R z = equivClass R y :=
    (Lemma_4_5_5_2 R h y z).ltr h7.right
  show X = Y from
    calc X

```

```

_ = equivClass R x := h4.symm
_ = equivClass R z := h8.symm
_ = equivClass R y := h9
_ = Y           := h5
done

lemma Theorem_4_5_4_part_3 {A : Type} (R : BinRel A) (h : equiv_rel R) :
  ∀ X ∈ mod A R, ¬empty X := by
  fix X : Set A
  assume h2 : X ∈ mod A R --Goal : ¬empty X
  define; double_neg      --Goal : ∃ (x : A), x ∈ X
  define at h2            --h2 : ∃ (x : A), equivClass R x = X
  obtain (x : A) (h3 : equivClass R x = X) from h2
  rewrite [←h3]
  show ∃ (x_1 : A), x_1 ∈ equivClass R x from
    Exists.intro x (Lemma_4_5_5_1 R h x)
  done

```

It's easy now to put everything together to prove Theorem 4.5.4.

```

theorem Theorem_4_5_4 {A : Type} (R : BinRel A) (h : equiv_rel R) :
  partition (mod A R) := And.intro (Theorem_4_5_4_part_1 R h)
    (And.intro (Theorem_4_5_4_part_2 R h) (Theorem_4_5_4_part_3 R h))

```

Theorem 4.5.4 shows that an equivalence relation on A determines a partition of A , namely $\text{mod } A \text{ } R$. Our next project will be to prove Theorem 4.5.6 in *HTPI*, which says that every partition of A arises in this way; that is, every partition is $\text{mod } A \text{ } R$ for some equivalence relation R . To prove this, we must show how to use a partition F to define an equivalence relation R for which $\text{mod } A \text{ } R = F$. The proof in *HTPI* defines the required equivalence relation R as a set of ordered pairs, but in Lean we will need to define it instead as a binary relation on A . Translating *HTPI*'s set-theoretic definition into Lean's notation for binary relations leads to the following definition:

```

def EqRelFromPart {A : Type} (F : Set (Set A)) (x y : A) : Prop :=
  ∃ X ∈ F, x ∈ X ∧ y ∈ X

```

In other words, `EqRelFromPart F` is the binary relation on A that is true of any two objects x and y of type A if and only if x and y belong to the same set in F . Our plan now is to show that if F is a partition of A , then `EqRelFromPart F` is an equivalence relation on A , and $\text{mod } A \text{ } (\text{EqRelFromPart } F) = F$.

4.5. Equivalence Relations

Once again, *HTPI* breaks the proof up by proving some lemmas first, and we will find it convenient to break the proof into even smaller pieces. We will leave the proofs of most of these lemmas as exercises for you.

```
lemma overlap_implies_equal {A : Type}
  (F : Set (Set A)) (h : partition F) :
  ∀ X ∈ F, ∀ Y ∈ F, ∀ (x : A), x ∈ X → x ∈ Y → X = Y := sorry

lemma Lemma_4_5_7_ref {A : Type} (F : Set (Set A)) (h : partition F) :
  reflexive (EqRelFromPart F) := sorry

lemma Lemma_4_5_7_symm {A : Type} (F : Set (Set A)) (h : partition F) :
  symmetric (EqRelFromPart F) := sorry

lemma Lemma_4_5_7_trans {A : Type} (F : Set (Set A)) (h : partition F) :
  transitive (EqRelFromPart F) := sorry
```

We can now put these pieces together to prove Lemma 4.5.7 in *HTPI*:

```
lemma Lemma_4_5_7 {A : Type} (F : Set (Set A)) (h : partition F) :
  equiv_rel (EqRelFromPart F) := And.intro (Lemma_4_5_7_ref F h)
    (And.intro (Lemma_4_5_7_symm F h) (Lemma_4_5_7_trans F h))
```

We need one more lemma before we can prove Theorem 4.5.6:

```
lemma Lemma_4_5_8 {A : Type} (F : Set (Set A)) (h : partition F) :
  ∀ X ∈ F, ∀ x ∈ X, equivClass (EqRelFromPart F) x = X := sorry
```

We are finally now ready to address Theorem 4.5.6. Here is the statement of the theorem:

```
theorem Theorem_4_5_6 {A : Type} (F : Set (Set A)) (h : partition F) :
  ∃ (R : BinRel A), equiv_rel R ∧ mod A R = F
```

Of course, the relation R that we will use to prove the theorem is `EqRelFromPart F`, so we could start the proof with the tactic `apply Exists.intro (EqRelFromPart F)`. But this means that the rest of the proof will involve many statements about the relation `EqRelFromPart F`. When a complicated object appears multiple times in a proof, it can make the proof a little easier to read if we give that object a name. We can do that by using a new tactic. The tactic `set R : BinRel A := EqRelFromPart F` introduces the new variable R into the tactic state. The variable R has type `BinRel A`, and it is definitionally equal to `EqRelFromPart F`. That means that, when necessary, Lean will fill in this definition of R . For example, one of our first steps will be to

apply Lemma_4_5_7 to F and h . The conclusion of that lemma is `equiv_rel (EqRelFromPart F)`, but Lean will recognize this as meaning the same thing as `equiv_rel R`. Here is the proof of the theorem:

```
theorem Theorem_4_5_6 {A : Type} (F : Set (Set A)) (h: partition F) :
  ∃ (R : BinRel A), equiv_rel R ∧ mod A R = F := by
  set R : BinRel A := EqRelFromPart F
  apply Exists.intro R           --Goal : equiv_rel R ∧ mod A R = F
  apply And.intro (Lemma_4_5_7 F h) --Goal : mod A R = F
  apply Set.ext
  fix X : Set A                 --Goal : X ∈ mod A R ↔ X ∈ F
  apply Iff.intro
  · -- (→)
    assume h2 : X ∈ mod A R      --Goal : X ∈ F
    define at h2                  --h2 : ∃ (x : A), equivClass R x = X
    obtain (x : A) (h3 : equivClass R x = X) from h2
    have h4 : x ∈ U₀ F := h.left x
    define at h4
    obtain (Y : Set A) (h5 : Y ∈ F ∧ x ∈ Y) from h4
    have h6 : equivClass R x = Y :=
      Lemma_4_5_8 F h Y h5.left x h5.right
    rewrite [←h3, h6]
    show Y ∈ F from h5.left
    done
  · -- (←)
    assume h2 : X ∈ F            --Goal : X ∈ mod A R
    have h3 : ¬empty X := h.right.right X h2
    define at h3; double_neg at h3 --h3 : ∃ (x : A), x ∈ X
    obtain (x : A) (h4 : x ∈ X) from h3
    define                         --Goal : ∃ (x : A), equivClass R x = X
    show ∃ (x : A), equivClass R x = X from
      Exists.intro x (Lemma_4_5_8 F h X h2 x h4)
    done
  done
```

Exercises

1. `lemma overlap_implies_equal {A : Type}`
`(F : Set (Set A)) (h : partition F) :`
 `∀ X ∈ F, ∀ Y ∈ F, ∀ (x : A), x ∈ X → x ∈ Y → X = Y := sorry`

2. `lemma Lemma_4_5_7_ref {A : Type} (F : Set (Set A)) (h : partition F) :
reflexive (EqRelFromPart F) := sorry`
3. `lemma Lemma_4_5_7_symm {A : Type} (F : Set (Set A)) (h : partition F) :
symmetric (EqRelFromPart F) := sorry`
4. `lemma Lemma_4_5_7_trans {A : Type} (F : Set (Set A)) (h : partition F) :
transitive (EqRelFromPart F) := sorry`
5. `lemma Lemma_4_5_8 {A : Type} (F : Set (Set A)) (h : partition F) :
 $\forall X \in F, \forall x \in X, \text{equivClass } (\text{EqRelFromPart } F) x = X := \text{sorry}$`
6. `lemma elt_mod_equiv_class_of_elt
{A : Type} (R : BinRel A) (h : equiv_rel R) :
 $\forall X \in \text{mod } A R, \forall x \in X, \text{equivClass } R x = X := \text{sorry}$`

The next three exercises use the following definitions:

```
def dot {A : Type} (F G : Set (Set A)) : Set (Set A) :=  
{Z : Set A |  $\neg\text{empty } Z \wedge \exists X \in F, \exists Y \in G, Z = X \cap Y\}$   
  
def conj {A : Type} (R S : BinRel A) (x y : A) : Prop :=  
R x y  $\wedge$  S x y
```

7. `theorem Exercise_4_5_20a {A : Type} (R S : BinRel A)
(h1 : equiv_rel R) (h2 : equiv_rel S) :
equiv_rel (conj R S) := sorry`
8. `theorem Exercise_4_5_20b {A : Type} (R S : BinRel A)
(h1 : equiv_rel R) (h2 : equiv_rel S) :
 $\forall (x : A), \text{equivClass } (\text{conj } R S) x =$
 $\text{equivClass } R x \cap \text{equivClass } S x := \text{sorry}$`
9. `theorem Exercise_4_5_20c {A : Type} (R S : BinRel A)
(h1 : equiv_rel R) (h2 : equiv_rel S) :
 $\text{mod } A (\text{conj } R S) = \text{dot } (\text{mod } A R) (\text{mod } A S) := \text{sorry}$`

The next exercise uses the following definition:

```
def equiv_mod (m x y : Int) : Prop := m  $|$  (x - y)  
  
10. theorem Theorem_4_5_10 :  $\forall (m : \text{Int}), \text{equiv\_rel } (\text{equiv\_mod } m) := \text{sorry}$ 
```

5 Functions

5.1. Functions

The first definition in Chapter 5 of *HTPI* says that if $F \subseteq A \times B$, then F is called a *function* from A to B if for every $a \in A$ there is exactly one $b \in B$ such that $(a, b) \in F$. The notation $F : A \rightarrow B$ means that F is a function from A to B . If F is a function from A to B and $a \in A$, then *HTPI* introduces the notation $F(a)$ for the unique $b \in B$ such that $(a, b) \in F$. Thus, if $F : A \rightarrow B$, $a \in A$, and $b \in B$, then $F(a) = b$ means the same thing as $(a, b) \in F$. We sometimes think of F as representing an operation that can be applied to an element a of A to produce a corresponding element $F(a)$ of B , and we call $F(a)$ the *value of F at a* , or the *result of applying F to a* .

This might remind you of the situation we faced in Chapter 4. If $R \subseteq A \times B$, $a \in A$, and $b \in B$, then Chapter 4 of *HTPI* uses the notation aRb to mean the same thing as $(a, b) \in R$. But in Lean, we found it necessary to change this notation. Instead of using *HTPI*'s notation aRb , we introduced the notation $R a b$, which we use when R has type $\text{Rel } A B$, a has type A , and b has type B . (The notation $(a, b) \in R$, in contrast, can be used only when R has type $\text{Set } (A \times B)$.) If R has type $\text{Rel } A B$, then we think of R as representing some relationship that might hold between a and b , and $R a b$ as the proposition saying that this relationship holds. And although R is not a set of ordered pairs, there is a corresponding set, $\text{extension } R$, of type $\text{Set } (A \times B)$, with the property that $(a, b) \in \text{extension } R$ if and only if $R a b$.

We will take a similar approach to functions in this chapter. For any types A and B , we introduce a new type $A \rightarrow B$. If f has type $A \rightarrow B$, then we think of f as representing some operation that can be applied to an object of type A to produce a corresponding object of type B . We will say that f is a *function* from A to B , and A is the *domain* of f . If a has type A , then we write $f a$ (with a space but no parentheses) for the result of applying the operation represented by f to the object a . Thus, if we have $f : A \rightarrow B$ and $a : A$, then $f a$ has type B . As with relations, if f has type $A \rightarrow B$, then f is not a set of ordered pairs. But there is a corresponding set of ordered pairs, which we will call the *graph* of f , whose elements are the ordered pairs (a, b) for which $f a = b$:

```
def graph {A B : Type} (f : A → B) : Set (A × B) :=
  {(a, b) : A × B | f a = b}
```

```
theorem graph_def {A B : Type} (f : A → B) (a : A) (b : B) :
  (a, b) ∈ graph f ↔ f a = b := by rfl
```

Every set of type `Set (A × B)` is the extension of some relation from `A` to `B`, but not every such set is the graph of a function from `A` to `B`. To be the graph of a function, it must have the property that was used to define functions in *HTPI*: each object of type `A` must be paired in the set with exactly one object of type `B`. Let's give this property a name:

```
def is_func_graph {A B : Type} (F : Set (A × B)) : Prop :=
  ∀ (x : A), ∃! (y : B), (x, y) ∈ F
```

And now we can say that the sets of type `Set (A × B)` that are graphs of functions from `A` to `B` are precisely the ones that have the property `is_func_graph`:

```
theorem func_from_graph {A B : Type} (F : Set (A × B)) :
  (∃ (f : A → B), graph f = F) ↔ is_func_graph F
```

We will ask you to prove the left-to-right direction of this theorem in the exercises. The proof of the right-to-left direction in Lean is tricky; it requires an idea that we won't introduce until Section 8.2. We'll give the proof then, but we'll go ahead and use the theorem in this chapter when we find it useful.

Section 5.1 of *HTPI* proves two theorems about functions. The first gives a convenient way of proving that two functions are equal (*HTPI* p. 232):

Theorem 5.1.4. *Suppose f and g are functions from A to B . If $\forall a \in A(f(a) = g(a))$, then $f = g$.*

The proof of this theorem in *HTPI* is based on the axiom of extensionality for sets. But in Lean, functions aren't sets of ordered pairs, so this method of proof won't work. Fortunately, Lean has a similar axiom of extensionality for functions. The axiom is called `funext`, and it proves Theorem 5.1.4.

```
theorem Theorem_5_1_4 {A B : Type} (f g : A → B) :
  (∀ (a : A), f a = g a) → f = g := funext
```

We saw previously that if we are trying to prove $X = Y$, where X and Y both have type `Set U`, then often the best first step is the tactic `apply Set.ext`, which converts the goal to $\forall (x : U), x \in X \leftrightarrow x \in Y$. Similarly, if we are trying to prove $f = g$, where f and g both have type `A → B`, then we will usually start with the tactic `apply funext`, which will convert the goal to $\forall (x : A), f x = g x$. By `Theorem_5_1_4`, this implies the original goal $f = g$. For example, here is a proof that if two functions have the same graph, then they are equal:

```
example {A B : Type} (f g : A → B) :
  graph f = graph g → f = g := by
  assume h1 : graph f = graph g --Goal : f = g
  apply funext --Goal : ∀ (x : A), f x = g x
  fix x : A
  have h2 : (x, f x) ∈ graph f := by
    define --Goal : f x = f x
    rfl
    done
  rewrite [h1] at h2 --h2 : (x, f x) ∈ graph g
  define at h2 --h2 : g x = f x
  show f x = g x from h2.symm
  done
```

The axiom of extensionality for sets says that a set is completely determined by its elements. This is what justifies our usual method of defining a set: we specify what its elements are, using notation like $\{0, 1, 2\}$ or $\{x : \text{Nat} \mid x < 3\}$. Similarly, the axiom of extensionality for functions says that a function is completely determined by its values, and therefore we can define a function by specifying its values. For instance, we can define a function from `Nat` to `Nat` by specifying, for any $n : \text{Nat}$, the result of applying the function to n . As an example of this, we could define the “squaring function” from `Nat` to `Nat` to be the function that, when applied to any $n : \text{Nat}$, produces the result $n ^ 2$. Here are two ways to define this function in Lean:

```
def square1 (n : Nat) : Nat := n ^ 2

def square2 : Nat → Nat := fun (n : Nat) => n ^ 2
```

The first of these definitions uses notation we have used before; it says that if n has type `Nat`, then the expression `square1 n` also has type `Nat`, and it is definitionally equal to $n ^ 2$. The second definition introduces new Lean notation. It says that `square2` has type `Nat → Nat`, and it defines it to be the function that, when applied to any n of type `Nat`, yields the result $n ^ 2$. Of course, this also means that `square2 n` is definitionally equal to $n ^ 2$. In general, the notation `fun (x : A) => ...` means “the function which, when applied to any x of type A , yields the result ...” The two definitions above are equivalent. You can ask Lean to confirm this, and try out the squaring function, as follows (the `#eval` command asks Lean to evaluate an expression):

```
example : square1 = square2 := by rfl

#eval square1 7 --Answer: 49
```

There is one more theorem in Section 5.1 of *HTPI*. Theorem 5.1.5 says that if f is a function from A to B and g is a function from B to C , then the composition of g and f is a function from A to C . To state this theorem in Lean, we will have to make adjustments for the differences between the treatment of functions in *HTPI* and Lean. In Chapter 4, we defined `comp S R` to be the composition of S and R , where R has type `Set (A × B)` and S had type `Set (B × C)`. But functions in Lean are not sets of ordered pairs, so we cannot apply the operation `comp` to them. We can, however, apply it to their graphs. So the theorem corresponding to Theorem 5.1.5 in *HTPI* is this:

```
theorem Theorem_5_1_5 {A B C : Type} (f : A → B) (g : B → C) :
  ∃ (h : A → C), graph h = comp (graph g) (graph f) := by
  set h : A → C := fun (x : A) => g (f x)
  apply Exists.intro h
  apply Set.ext
  fix (a, c) : A × C
  apply Iff.intro
  · -- Proof that (a, c) ∈ graph h → (a, c) ∈ comp (graph g) (graph f)
    assume h1 : (a, c) ∈ graph h
    define at h1 --h1 : h a = c
    define      --Goal : ∃ (x : B), (a, x) ∈ graph f ∧ (x, c) ∈ graph g
    apply Exists.intro (f a)
    apply And.intro
    · -- Proof that (a, f a) ∈ graph f
      define
      rfl
      done
    · -- Proof that (f a, c) ∈ graph g
      define
      show g (f a) = c from h1
      done
    done
  · -- Proof that (a, c) ∈ comp (graph g) (graph f) → (a, c) ∈ graph h
    assume h1 : (a, c) ∈ comp (graph g) (graph f)
    define      --Goal : h a = c
    define at h1 --h1 : ∃ (x : B), (a, x) ∈ graph f ∧ (x, c) ∈ graph g
    obtain (b : B) (h2 : (a, b) ∈ graph f ∧ (b, c) ∈ graph g) from h1
    have h3 : (a, b) ∈ graph f := h2.left
    have h4 : (b, c) ∈ graph g := h2.right
    define at h3      --h3 : f a = b
    define at h4      --h4 : g b = c
    rewrite [←h3] at h4      --h4 : g (f a) = c
    show h a = c from h4
    done
```

done

Notice that the proof of `Theorem_5_1_5` begins by defining the function `h` for which `graph h = comp (graph g) (graph f)`. The definition says that for all x of type A , $h x = g (f x)$. This function `h` is called the *composition* of `g` and `f`, and it is denoted $g \circ f$. (To type \circ in VS Code, type `\comp` or `\circ`.) In other words, $g \circ f$ has type $A \rightarrow C$, and for all x of type A , $(g \circ f) x$ is definitionally equal to $g (f x)$. In *HTPI*, functions are sets of ordered pairs, and the operation of composition of functions is literally the same as the operation `comp` that we used in Chapter 4. But in Lean, we distinguish among functions, relations, and sets of ordered pairs, so all we can say is that the operation of composition of functions corresponds to the operation `comp` from Chapter 4. The correspondence is that, as shown in the proof of `Theorem_5_1_5`, if $h = g \circ f$, then `graph h = comp (graph g) (graph f)`.

We saw in part 4 of Theorem 4.2.5 that composition of relations is associative. Composition of functions is also associative. In fact, if $f : A \rightarrow B$, $g : B \rightarrow C$, and $h : C \rightarrow D$, then $h \circ (g \circ f)$ and $(h \circ g) \circ f$ are definitionally equal, since they both mean the same thing as `fun (x : A) => h (g (f a))`. As a result, the tactic `rfl` proves the associativity of composition of functions:

```
example {A B C D : Type} (f : A → B) (g : B → C) (h : C → D) :
  h ∘ (g ∘ f) = (h ∘ g) ∘ f := by rfl
```

HTPI defines the identity function on a set A to be the function i_A from A to A such that $\forall x \in A (i_A(x) = x)$, and Exercise 9 from Section 4.3 of *HTPI* implies that if $f : A \rightarrow B$, then $f \circ i_A = f$ and $i_B \circ f = f$. We say, therefore, that the identity functions are *identity elements* for composition of functions. Similarly, in Lean, for each type `A` there is an identity function from `A` to `A`. This identity function is denoted `id`; there is no need to specify `A` in the notation, because `A` is an implicit argument to `id`. Thus, when you use `id` to denote an identity function, Lean will figure out what type `A` to use as the domain of the function. (If, for some reason, you want to specify that the domain is some type `A`, you can write `@id A` instead of `id`.) For any x , of any type, `id x` is definitionally equal to x , and as a result the proof that `id` is an identity element for composition of functions can also be done with the `rfl` tactic:

```
example {A B : Type} (f : A → B) : f ∘ id = f := by rfl
```

```
example {A B : Type} (f : A → B) : id ∘ f = f := by rfl
```

Exercises

1. `theorem func_from_graph_ltr {A B : Type} (F : Set (A × B)) :
 (exists (f : A → B), graph f = F) → is_func_graph F := sorry`

2. `theorem Exercise_5_1_13a`
 $\{A B C : \text{Type}\} (R : \text{Set}(A \times B)) (S : \text{Set}(B \times C)) (f : A \rightarrow C)$
 $(h1 : \forall (b : B), b \in \text{Ran } R \wedge b \in \text{Dom } S) (h2 : \text{graph } f = \text{comp } S R) :$
`is_func_graph S := sorry`
3. `theorem Exercise_5_1_14a`
 $\{A B : \text{Type}\} (f : A \rightarrow B) (R : \text{BinRel } A) (S : \text{BinRel } B)$
 $(h : \forall (x y : A), R x y \leftrightarrow S(f x) (f y)) :$
`reflexive S → reflexive R := sorry`

4. Here is a putative theorem:

Theorem?. Suppose $f : A \rightarrow B$, R is a binary relation on A , and S is the binary relation on B defined as follows:

$$\forall x \in B \forall y \in B (xSy \leftrightarrow \exists u \in A \exists v \in A (f(u) = x \wedge f(v) = y \wedge uRv)).$$

If R is reflexive then S is reflexive.

Is the theorem correct? Try to prove it in Lean. If you can't prove it, see if you can find a counterexample.

```
--You might not be able to complete this proof
theorem Exercise_5_1_15a
  {A B : Type} (f : A → B) (R : BinRel A) (S : BinRel B)
  (h : ∀ (x y : B), S x y ↔ ∃ (u v : A), f u = x ∧ f v = y ∧ R u v) :
  reflexive R → reflexive S := sorry
```

5. Here is a putative theorem with an incorrect proof:

Theorem?. Suppose $f : A \rightarrow B$, R is a binary relation on A , and S is the binary relation on B defined as follows:

$$\forall x \in B \forall y \in B (xSy \leftrightarrow \exists u \in A \exists v \in A (f(u) = x \wedge f(v) = y \wedge uRv)).$$

If R is transitive then S is transitive.

Incorrect Proof. Suppose R is transitive. Let x , y , and z be arbitrary elements of B . Assume that xSy and ySz . By the definition of S , this means that there are u , v , and w in A such that $f(u) = x$, $f(v) = y$, $f(w) = z$, uRv , and vRw . Since R is transitive, it follows that uRw . Since $f(u) = x$, $f(w) = z$, and uRw , xSz . Therefore S is transitive. \square

Find the mistake in the proof by attempting to write the proof in Lean. Is the theorem correct?

```
--You might not be able to complete this proof
theorem Exercise_5_1_15c
{A B : Type} (f : A → B) (R : BinRel A) (S : BinRel B)
(h : ∀ (x y : B), S x y ↔ ∃ (u v : A), f u = x ∧ f v = y ∧ R u v) :
transitive R → transitive S := sorry
```

6. `theorem Exercise_5_1_16b`
 $\{A B : \text{Type}\} (R : \text{BinRel } B) (S : \text{BinRel } (A \rightarrow B))$
 $(h : \forall (f g : A \rightarrow B), S f g \leftrightarrow \forall (x : A), R (f x) (g x)) :$
 $\text{symmetric } R \rightarrow \text{symmetric } S := \text{sorry}$
7. `theorem Exercise_5_1_17a {A : Type} (f : A → A) (a : A)`
 $(h : \forall (x : A), f x = a) : \forall (g : A \rightarrow A), f \circ g = f := \text{sorry}$
8. `theorem Exercise_5_1_17b {A : Type} (f : A → A) (a : A)`
 $(h : \forall (g : A \rightarrow A), f \circ g = f) :$
 $\exists (y : A), \forall (x : A), f x = y := \text{sorry}$

5.2. One-to-One and Onto

Section 5.2 of *HTPI* introduces two important properties that a function might have. A function $f : A \rightarrow B$ is called *onto* if for every b of type B there is at least one a of type A such that $f a = b$:

```
def onto {A B : Type} (f : A → B) : Prop :=
∀ (y : B), ∃ (x : A), f x = y
```

It is called *one-to-one* if there do *not* exist distinct a_1 and a_2 of type A such that $f a_1 = f a_2$. This phrasing of the definition makes it clear what is at issue: Are there distinct objects in the domain to which the function assigns the same value? But it is a negative statement, and that would make it difficult to work with it in proofs. Fortunately, it is not hard to rephrase the definition as an equivalent positive statement, using quantifier negation, De Morgan, and conditional laws. The resulting equivalent positive statement is given in Theorem 5.2.3 of *HTPI*, and we take it as our official definition of *one_to_one* in Lean:

```
def one_to_one {A B : Type} (f : A → B) : Prop :=
∀ (x1 x2 : A), f x1 = f x2 → x1 = x2
```

5.2. One-to-One and Onto

There is only one more theorem about these properties in Section 5.2 of *HTPI*. It says that a composition of one-to-one functions is one-to-one, and a composition of onto functions is onto. It is straightforward to carry out these proofs in Lean by simply applying the definitions of the relevant concepts.

```
theorem Theorem_5_2_5_1 {A B C : Type} (f : A → B) (g : B → C) :
  one_to_one f → one_to_one g → one_to_one (g ∘ f) := by
  assume h1 : one_to_one f
  assume h2 : one_to_one g
  define at h1 --h1 : ∀ (x1 x2 : A), f x1 = f x2 → x1 = x2
  define at h2 --h2 : ∀ (x1 x2 : B), g x1 = g x2 → x1 = x2
  define --Goal : ∀ (x1 x2 : A), (g ∘ f) x1 = (g ∘ f) x2 → x1 = x2
  fix a1 : A
  fix a2 : A --Goal : (g ∘ f) a1 = (g ∘ f) a2 → a1 = a2
  define : (g ∘ f) a1; define : (g ∘ f) a2
    --Goal : g (f a1) = g (f a2) → a1 = a2
  assume h3 : g (f a1) = g (f a2)
  have h4 : f a1 = f a2 := h2 (f a1) (f a2) h3
  show a1 = a2 from h1 a1 a2 h4
  done
```

Notice that the tactic `define : (g ∘ f) a1` replaces `(g ∘ f) a1` with its definition, `g (f a1)`. As usual, this step isn't really needed—Lean will apply the definition on its own when necessary, without being told. But using this tactic makes the proof easier to read. Also, notice that `define : g ∘ f` produces a result that is much less useful. As we have observed before, the `define` tactic works best when applied to a complete expression, rather than just a part of an expression.

An alternative way to apply the definition of composition of functions is to prove a lemma that can be used in the `rewrite` tactic. We try out this approach for the second part of the theorem.

```
lemma comp_def {A B C : Type} (f : A → B) (g : B → C) (x : A) :
  (g ∘ f) x = g (f x) := by rfl

theorem Theorem_5_2_5_2 {A B C : Type} (f : A → B) (g : B → C) :
  onto f → onto g → onto (g ∘ f) := by
  assume h1 : onto f
  assume h2 : onto g
  define at h1 --h1 : ∀ (y : B), ∃ (x : A), f x = y
  define at h2 --h2 : ∀ (y : C), ∃ (x : B), g x = y
  define --Goal : ∀ (y : C), ∃ (x : A), (g ∘ f) x = y
  fix c : C
```

```

obtain (b : B) (h3 : g b = c) from h2 c
obtain (a : A) (h4 : f a = b) from h1 b
apply Exists.intro a --Goal : (g ∘ f) a = c
rewrite [comp_def] --Goal : g (f a) = c
rewrite [←h4] at h3
show g (f a) = c from h3
done

```

Exercises

1. `theorem Exercise_5_2_10a {A B C : Type} (f: A → B) (g : B → C) :
onto (g ∘ f) → onto g := sorry`
2. `theorem Exercise_5_2_10b {A B C : Type} (f: A → B) (g : B → C) :
one_to_one (g ∘ f) → one_to_one f := sorry`
3. `theorem Exercise_5_2_11a {A B C : Type} (f: A → B) (g : B → C) :
onto f → ¬(one_to_one g) → ¬(one_to_one (g ∘ f)) := sorry`
4. `theorem Exercise_5_2_11b {A B C : Type} (f: A → B) (g : B → C) :
¬(onto f) → one_to_one g → ¬(onto (g ∘ f)) := sorry`
5. `theorem Exercise_5_2_12 {A B : Type} (f : A → B) (g : B → Set A)
(h : ∀ (b : B), g b = {a : A | f a = b}) :
onto f → one_to_one g := sorry`
6. `theorem Exercise_5_2_16 {A B C : Type}
(R : Set (A × B)) (S : Set (B × C)) (f : A → C) (g : B → C)
(h1 : graph f = comp S R) (h2 : graph g = S) (h3 : one_to_one g) :
is_func_graph R := sorry`
7. `theorem Exercise_5_2_17a
{A B : Type} (f : A → B) (R : BinRel A) (S : BinRel B)
(h1 : ∀ (x y : B), S x y ↔ ∃ (u v : A), f u = x ∧ f v = y ∧ R u v)
(h2 : onto f) : reflexive R → reflexive S := sorry`
8. `theorem Exercise_5_2_17b
{A B : Type} (f : A → B) (R : BinRel A) (S : BinRel B)
(h1 : ∀ (x y : B), S x y ↔ ∃ (u v : A), f u = x ∧ f v = y ∧ R u v)
(h2 : one_to_one f) : transitive R → transitive S := sorry`
9. `theorem Exercise_5_2_21a {A B C : Type} (f : B → C) (g h : A → B)
(h1 : one_to_one f) (h2 : f ∘ g = f ∘ h) : g = h := sorry`

```
10. theorem Exercise_5_2_21b {A B C : Type} (f : B → C) (a : A)
    (h1 : ∀ (g h : A → B), f ∘ g = f ∘ h → g = h) :
    one_to_one f := sorry
```

5.3. Inverses of Functions

Section 5.3 of *HTPI* is motivated by the following question: If f is a function from A to B , is f^{-1} a function from B to A ? Here is the first theorem in that section (*HTPI* p. 250):

Theorem 5.3.1. *Suppose $f : A \rightarrow B$. If f is one-to-one and onto, then $f^{-1} : B \rightarrow A$.*

Of course, we will have to rephrase this theorem slightly to prove it in Lean. If f has type $A \rightarrow B$, then the inverse operation `inv` cannot be applied to f , but it can be applied to `graph f`. So we must rephrase the theorem like this:

```
theorem Theorem_5_3_1 {A B : Type}
  (f : A → B) (h1 : one_to_one f) (h2 : onto f) :
  ∃ (g : B → A), graph g = inv (graph f)
```

To prove this theorem, we will use the theorem `func_from_graph` that was stated in Section 5.1. We can remind ourselves of what that theorem says by using the command `#check @func_from_graph`, which gives the result:

```
@func_from_graph : ∀ {A B : Type} (F : Set (A × B)),
  (∃ (f : A → B), graph f = F) ↔ is_func_graph F
```

This means that, in the context of the proof of `Theorem_5_3_1`, `func_from_graph (inv (graph f))` is a proof of the statement

```
∃ (g : B → A), graph g = inv (graph f) ↔ is_func_graph (inv (graph f)).
```

Therefore the tactic `rewrite [func_from_graph (inv (graph f))]` will change the goal to `is_func_graph (inv (graph f))`. In fact, we can just use `rewrite [func_from_graph]`, and Lean will figure out how to apply the theorem to rewrite the goal. The rest of the proof is straightforward.

```
theorem Theorem_5_3_1 {A B : Type}
  (f : A → B) (h1 : one_to_one f) (h2 : onto f) :
  ∃ (g : B → A), graph g = inv (graph f) := by
  rewrite [func_from_graph] --Goal : is_func_graph (inv (graph f))
  define          --Goal : ∀ (x : B), ∃! (y : A), (x, y) ∈ inv (graph f)
```

```

fix b : B
exists_unique
• -- Existence
  define at h2          --h2 : ∀ (y : B), ∃ (x : A), f x = y
  obtain (a : A) (h4 : f a = b) from h2 b
  apply Exists.intro a  --Goal : (b, a) ∈ inv (graph f)
  define                 --Goal : f a = b
  show f a = b from h4
  done
• -- Uniqueness
  fix a1 : A; fix a2 : A
  assume h3 : (b, a1) ∈ inv (graph f)
  assume h4 : (b, a2) ∈ inv (graph f) --Goal : a1 = a2
  define at h3          --h3 : f a1 = b
  define at h4          --h4 : f a2 = b
  rewrite [←h4] at h3  --h3 : f a1 = f a2
  define at h1          --h1 : ∀ (x1 x2 : A), f x1 = f x2 → x1 = x2
  show a1 = a2 from h1 a1 a2 h3
  done
done

```

Suppose, now, that we have $f : A \rightarrow B$, $g : B \rightarrow A$, and $\text{graph } g = \text{inv}(\text{graph } f)$, as in Theorem_5_3_1. What can we say about the relationship between f and g ? One answer is that $g \circ f = \text{id}$ and $f \circ g = \text{id}$, as shown in Theorem 5.3.2 of *HTPI*. We'll prove one of these facts, and leave the other as an exercise for you.

```

theorem Theorem_5_3_2_1 {A B : Type} (f : A → B) (g : B → A)
  (h1 : graph g = inv (graph f)) : g ∘ f = id := by
  apply funext           --Goal : ∀ (x : A), (g ∘ f) x = id x
  fix a : A              --Goal : (g ∘ f) a = id a
  have h2 : (f a, a) ∈ graph g := by
    rewrite [h1]          --Goal : (f a, a) ∈ inv (graph f)
    define                 --Goal : f a = f a
    rfl
    done
  define at h2          --h2 : g (f a) = a
  show (g ∘ f) a = id a from h2
  done

theorem Theorem_5_3_2_2 {A B : Type} (f : A → B) (g : B → A)
  (h1 : graph g = inv (graph f)) : f ∘ g = id := sorry

```

5.3. Inverses of Functions

Combining the theorems above, we have shown that if f is one-to-one and onto, then there is a function g such that $g \circ f = \text{id}$ and $f \circ g = \text{id}$. In fact, the converse is true as well: if such a function g exists, then f must be one-to-one and onto. Again, we'll prove one statement and leave the second as an exercise.

```
theorem Theorem_5_3_3_1 {A B : Type} (f : A → B) (g : B → A)
  (h1 : g ∘ f = id) : one_to_one f := by
  define          --Goal : ∀ (x1 x2 : A), f x1 = f x2 → x1 = x2
  fix a1 : A; fix a2 : A
  assume h2 : f a1 = f a2
  show a1 = a2 from
    calc a1
      _ = id a1 := by rfl
      _ = (g ∘ f) a1 := by rw [h1]
      _ = g (f a1) := by rfl
      _ = g (f a2) := by rw [h2]
      _ = (g ∘ f) a2 := by rfl
      _ = id a2 := by rw [h1]
      _ = a2 := by rfl
  done

theorem Theorem_5_3_3_2 {A B : Type} (f : A → B) (g : B → A)
  (h1 : f ∘ g = id) : onto f := sorry
```

We can combine the theorems above to show that if we have $f : A \rightarrow B$, $g : B \rightarrow A$, $g \circ f = \text{id}$, and $f \circ g = \text{id}$, then $\text{graph } g$ must be the inverse of $\text{graph } f$. Compare the proof below to the proof of Theorem 5.3.5 in *HTPI*.

```
theorem Theorem_5_3_5 {A B : Type} (f : A → B) (g : B → A)
  (h1 : g ∘ f = id) (h2 : f ∘ g = id) : graph g = inv (graph f) := by
  have h3 : one_to_one f := Theorem_5_3_3_1 f g h1
  have h4 : onto f := Theorem_5_3_3_2 f g h2
  obtain (g' : B → A) (h5 : graph g' = inv (graph f))
    from Theorem_5_3_1 f h3 h4
  have h6 : g' ∘ f = id := Theorem_5_3_2_1 f g' h5
  have h7 : g = g' :=
    calc g
      _ = id ∘ g := by rfl
      _ = (g' ∘ f) ∘ g := by rw [h6]
      _ = g' ∘ (f ∘ g) := by rfl
      _ = g' ∘ id := by rw [h2]
      _ = g' := by rfl
```

```
rewrite [←h7] at h5
show graph g = inv (graph f) from h5
done
```

Exercises

1. `theorem Theorem_5_3_2_2 {A B : Type} (f : A → B) (g : B → A)
(h1 : graph g = inv (graph f)) : f ∘ g = id := sorry`
2. `theorem Theorem_5_3_3_2 {A B : Type} (f : A → B) (g : B → A)
(h1 : f ∘ g = id) : onto f := sorry`
3. `theorem Exercise_5_3_11a {A B : Type} (f : A → B) (g : B → A) :
one_to_one f → f ∘ g = id → graph g = inv (graph f) := sorry`
4. `theorem Exercise_5_3_11b {A B : Type} (f : A → B) (g : B → A) :
onto f → g ∘ f = id → graph g = inv (graph f) := sorry`
5. `theorem Exercise_5_3_14a {A B : Type} (f : A → B) (g : B → A)
(h : f ∘ g = id) : ∀ x ∈ Ran (graph g), g (f x) = x := sorry`
6. `theorem Exercise_5_3_18 {A B C : Type} (f : A → C) (g : B → C)
(h1 : one_to_one g) (h2 : onto g) :
∃ (h : A → B), g ∘ h = f := sorry`

The next two exercises will use the following definition:

```
def conjugate (A : Type) (f1 f2 : A → A) : Prop :=
∃ (g g' : A → A), (f1 = g' ∘ f2 ∘ g) ∧ (g ∘ g' = id) ∧ (g' ∘ g = id)
```

7. `theorem Exercise_5_3_17a {A : Type} : symmetric (conjugate A) := sorry`
8. `theorem Exercise_5_3_17b {A : Type} (f1 f2 : A → A)
(h1 : conjugate A f1 f2) (h2 : ∃ (a : A), f1 a = a) :
∃ (a : A), f2 a = a := sorry`

5.4. Closures

Suppose we have $f : A \rightarrow A$ and $C : \text{Set } A$. We say that C is *closed* under f if the value of f at any element of C is again an element of C :

```
def closed {A : Type} (f : A → A) (C : Set A) : Prop := ∀ x ∈ C, f x ∈ C
```

According to this definition, `closed f C` means that C is closed under f . Sometimes, if we have a set B of type `Set A` that is not closed under f , we are interested in adding more elements to the set to make it closed. The *closure* of B under f is the smallest set containing B that is closed under f . That is, it is the smallest element of $\{D : \text{Set } A \mid B ⊆ D \wedge \text{closed } f D\}$, where we use the subset partial ordering on `Set A` to determine which element is smallest. We will write `closure f B C` to mean that the closure of B under f is C . We can define this as follows:

```
def closure {A : Type} (f : A → A) (B C : Set A) : Prop :=
smallestElt (sub A) C {D : Set A | B ⊆ D ∧ closed f D}
```

We know that smallest elements, when they exist, are unique, so it makes sense to talk about *the* closure of B under f . But not every set has a smallest element. Does every set have a closure? Theorem 5.4.5 in *HTPI* says that the answer is yes. The idea behind the proof is that, for any family of sets F , if F has a smallest element under the subset partial order, then that smallest element is equal to $\cap_0 F$. (We'll ask you to prove this in the exercises.)

```
theorem Theorem_5_4_5 {A : Type} (f : A → A) (B : Set A) :
∃ (C : Set A), closure f B C := by
set F : Set (Set A) := {D : Set A | B ⊆ D ∧ closed f D}
set C : Set A := ⋂₀ F
apply Exists.intro C      --Goal : closure f B C
define                  --Goal : C ∈ F ∧ ∀ x ∈ F, C ⊆ x
apply And.intro
· -- Proof that C ∈ F
  define                  --Goal : B ⊆ C ∧ closed f C
  apply And.intro
  · -- Proof that B ⊆ C
    fix a : A
    assume h1 : a ∈ B      --Goal : a ∈ C
    define                  --Goal : ∀ t ∈ F, a ∈ t
    fix D : Set A
    assume h2 : D ∈ F
    define at h2           --h2 : B ⊆ D ∧ closed f D
    show a ∈ D from h2.left h1
```

```

done
• -- Proof that C is closed under f
define                      --Goal : ∀ x ∈ C, f x ∈ C
fix a : A
assume h1 : a ∈ C          --Goal : f a ∈ C
define                      --Goal : ∀ t ∈ F, f a ∈ t
fix D : Set A
assume h2 : D ∈ F          --Goal : f a ∈ D
define at h1                --h1 : ∀ t ∈ F, a ∈ t
have h3 : a ∈ D := h1 D h2
define at h2                --h2 : B ⊆ D ∧ closed f D
have h4 : closed f D := h2.right
define at h4                --h4 : ∀ x ∈ D, f x ∈ D
show f a ∈ D from h4 a h3
done
done
done
• -- Proof that C is smallest
fix D : Set A
assume h1 : D ∈ F          --Goal : sub A C D
define
fix a : A
assume h2 : a ∈ C          --Goal : a ∈ D
define at h2                --h2 : ∀ t ∈ F, a ∈ t
show a ∈ D from h2 D h1
done
done
done

```

The idea of the closure of a set under a function can also be applied to functions of two variables. One way to represent a function of two variables on a type A would be to use a function g of type $(A \times A) \rightarrow A$. If a and b have type A , then (a, b) has type $A \times A$, and the result of applying the function g to the pair of values a and b would be written $g(a, b)$.

However, there is another way to represent a function of two variables that turns out to be more convenient in Lean. Suppose f has type $A \rightarrow A \rightarrow A$. As with the arrow used in conditional propositions, the arrow for function types groups to the right, so $A \rightarrow A \rightarrow A$ means $A \rightarrow (A \rightarrow A)$. Thus, if a has type A , then $f a$ has type $A \rightarrow A$. In other words, $f a$ is a function from A to A , and therefore if b has type A then $f a b$ has type A . The upshot is that if f is followed by two objects of type A , then the resulting expression has type A , so f can be thought of as a function that applies to a pair of objects of type A and gives a value of type A .

For example, we can think of addition of integers as a function of two variables. Here are three ways to define this function in Lean.

```
def plus (m n : Int) : Int := m + n

def plus' : Int → Int → Int := fun (m n : Int) => m + n

def plus'' : Int → Int → Int := fun (m : Int) => (fun (n : Int) => m + n)
```

The third definition matches the description above most closely: `plus''` is a function that, when applied to an integer `m`, produces a new function `plus'' m : Int → Int`. The function `plus'' m` is defined to be the function that, when applied to an integer `n`, produces the value `m + n`. In other words, `plus'' m n = m + n`. The first two definitions are more convenient ways of defining exactly the same function. Let's have Lean confirm this, and try out the function:

```
example : plus = plus'' := by rfl

example : plus' = plus'' := by rfl

#eval plus 3 2      --Answer: 5
```

There are two reasons why this way of representing functions of two variables in Lean is more convenient. First, it saves us the trouble of grouping the arguments of the function together into an ordered pair before applying the function. If we have `f : A → A → A` and `a b : A`, then to apply the function `f` to the arguments `a` and `b` we can just write `f a b`. Second, it allows for the possibility of “partially applying” the function `f`. The expression `f a` is meaningful, and denotes the function that, when applied to any `b : A`, produces the result `f a b`. For example, if `m` is an integer, then `plus m` denotes the function that, when applied to an integer `n`, produces the result `m + n`. We might call `plus m` the “add to `m`” function.

We have actually been using these ideas for a long time. In Chapter 3, we introduced the type `Pred U` of predicates applying to objects of type `U`, but we did not explain how such predicates are represented internally in Lean. In fact, `Pred U` is defined to be the type `U → Prop`, so if `P` has type `Pred U`, then `P` is a function from `U` to `Prop`, and if `x` has type `U`, then the proposition `P x` is the result of applying the function `P` to `x`. Similarly, `Rel A B` stands for `A → B → Prop`, so if `R` has type `Rel A B`, then `R` is a function of two variables, one of type `A` and one of type `B`. Earlier in this section, we defined `closed f c` to be the proposition asserting that `c` is closed under `f`. This means that `closed` is a function of two variables, the first a function `f` of type `A → A` and the second a set `c` of type `Set A` (where the type `A` is an implicit argument of `closed`). But that means that the partial application `closed f` denotes a function from `Set A` to `Prop`. In other words, `closed f` is a predicate applying to sets of type `Set A`; we could think of it as the “is closed under `f`” predicate. Similarly, in Section 4.4 we defined `sub` to be a function of three variables: if `A` is a type and `X` and `Y` have type `Set A`, then `sub A X Y` is the proposition `X ⊂ Y`. Since then, we have used the partial application `sub A`, which is the subset relation on `Set A`. For example, we used it earlier in this section in the definition of `closure`.

5.4. Closures

Returning to the subject of closures, here's how we can extend the idea of closures to functions of two variables:

```
def closed2 {A : Type} (f : A → A → A) (C : Set A) : Prop :=
  ∀ x ∈ C, ∀ y ∈ C, f x y ∈ C

def closure2 {A : Type} (f : A → A → A) (B C : Set A) : Prop :=
  smallestElt (sub A) C {D : Set A | B ⊆ D ∧ closed2 f D}
```

We will leave it as an exercise for you to prove that closures under functions of two variables also exist.

```
theorem Theorem_5_4_9 {A : Type} (f : A → A → A) (B : Set A) :
  ∃ (C : Set A), closure2 f B C := sorry
```

Exercises

1.

```
example {A : Type} (F : Set (Set A)) (B : Set A) :
  smallestElt (sub A) B F → B = ⋂₀ F := sorry
```

2. If B has type $\text{Set } A$, then $\text{complement } B$ is the set $\{a : A \mid a \notin B\}$. Thus, for any a of type A , $a \in \text{complement } B$ if and only if $a \notin B$:

```
def complement {A : Type} (B : Set A) : Set A := {a : A | a ∉ B}
```

Use this definition to prove the following theorem:

```
theorem Exercise_5_4_7 {A : Type} (f g : A → A) (C : Set A)
  (h1 : f ∘ g = id) (h2 : closed f C) : closed g (complement C) := sorry
```

3.

```
theorem Exercise_5_4_9a {A : Type} (f : A → A) (C1 C2 : Set A)
  (h1 : closed f C1) (h2 : closed f C2) : closed f (C1 ∪ C2) := sorry
```
4.

```
theorem Exercise_5_4_10a {A : Type} (f : A → A) (B1 B2 C1 C2 : Set A)
  (h1 : closure f B1 C1) (h2 : closure f B2 C2) :
  B1 ⊆ B2 → C1 ⊆ C2 := sorry
```
5.

```
theorem Exercise_5_4_10b {A : Type} (f : A → A) (B1 B2 C1 C2 : Set A)
  (h1 : closure f B1 C1) (h2 : closure f B2 C2) :
  closure f (B1 ∪ B2) (C1 ∪ C2) := sorry
```

```
6. theorem Theorem_5_4_9 {A : Type} (f : A → A → A) (B : Set A) :
  ∃ (C : Set A), closure2 f B C := sorry
```

7. Suppose we define a set to be closed under a family of functions if it is closed under all of the functions in the family. Of course, the closure of a set B under a family of functions is the smallest set containing B that is closed under the family.

```
def closed_family {A : Type} (F : Set (A → A)) (C : Set A) : Prop :=
  ∀ f ∈ F, closed f C

def closure_family {A : Type} (F : Set (A → A)) (B C : Set A) : Prop :=
  smallestElt (sub A) C {D : Set A | B ⊆ D ∧ closed_family F D}
```

Prove that the closure of a set under a family of functions always exists:

```
theorem Exercise_5_4_13a {A : Type} (F : Set (A → A)) (B : Set A) :
  ∃ (C : Set A), closure_family F B C := sorry
```

5.5. Images and Inverse Images: A Research Project

Section 5.5 of *HTPI* introduces two new definitions (*HTPI* p. 268). Suppose $f : A \rightarrow B$. If $X \subseteq A$, then the *image* of X under f is the set $f(X)$ defined as follows:

$$f(X) = \{f(x) \mid x \in X\} = \{b \in B \mid \exists x \in X (f(x) = b)\}.$$

If $Y \subseteq B$, then the *inverse image* of Y under f is the set $f^{-1}(Y)$ defined as follows:

$$f^{-1}(Y) = \{a \in A \mid f(a) \in Y\}.$$

Here are definitions of these concepts in Lean:

```
def image {A B : Type} (f : A → B) (X : Set A) : Set B :=
  {f x | x ∈ X}

def inverse_image {A B : Type} (f : A → B) (Y : Set B) : Set A :=
  {a : A | f a ∈ Y}

--The following theorems illustrate the meaning of these definitions:
theorem image_def {A B : Type} (f : A → B) (X : Set A) (b : B) :
  b ∈ image f X ↔ ∃ x ∈ X, f x = b := by rfl

theorem inverse_image_def {A B : Type} (f : A → B) (Y : Set B) (a : A) :
  a ∈ inverse_image f Y ↔ f a ∈ Y := by rfl
```

5.5. Images and Inverse Images: A Research Project

It is natural to wonder how these concepts interact with familiar operations on sets. *HTPI* gives an example of such an interaction in Theorem 5.5.2. The theorem makes two assertions. Here are proofs of the two parts of the theorem in Lean.

```

theorem Theorem_5_5_2_1 {A B : Type} (f : A → B) (W X : Set A) :
  image f (W ∩ X) ⊆ image f W ∩ image f X := by
  fix y : B
  assume h1 : y ∈ image f (W ∩ X) --Goal : y ∈ image f W ∩ image f X
  define at h1                   --h1 : ∃ x ∈ W ∩ X, f x = y
  obtain (x : A) (h2 : x ∈ W ∩ X ∧ f x = y) from h1
  define : x ∈ W ∩ X at h2      --h2 : (x ∈ W ∧ x ∈ X) ∧ f x = y
  apply And.intro
  · -- Proof that y ∈ image f W
    define                               --Goal : ∃ x ∈ W, f x = y
    show ∃ (x : A), x ∈ W ∧ f x = y from
      Exists.intro x (And.intro h2.left.left h2.right)
    done
  · -- Proof that y ∈ image f X
    show y ∈ image f X from
      Exists.intro x (And.intro h2.left.right h2.right)
    done
  done

theorem Theorem_5_5_2_2 {A B : Type} (f : A → B) (W X : Set A)
  (h1 : one_to_one f) : image f (W ∩ X) = image f W ∩ image f X := by
  apply Set.ext
  fix y : B           --Goal : y ∈ image f (W ∩ X) ↔ y ∈ image f W ∩ image f X
  apply Iff.intro
  · -- (→)
    assume h2 : y ∈ image f (W ∩ X)
    show y ∈ image f W ∩ image f X from Theorem_5_5_2_1 f W X h2
    done
  · -- (←)
    assume h2 : y ∈ image f W ∩ image f X --Goal : y ∈ image f (W ∩ X)
    define at h2                   --h2 : y ∈ image f W ∧ y ∈ image f X
    rewrite [image_def, image_def] at h2
    --h2 : (∃ x ∈ W, f x = y) ∧ ∃ x ∈ X, f x = y
    obtain (x1 : A) (h3 : x1 ∈ W ∧ f x1 = y) from h2.left
    obtain (x2 : A) (h4 : x2 ∈ X ∧ f x2 = y) from h2.right
    have h5 : f x2 = y := h4.right
    rewrite [←h3.right] at h5 --h5 : f x2 = f x1
    define at h1                   --h1 : ∀ (x1 x2 : A), f x1 = f x2 → x1 = x2
    have h6 : x2 = x1 := h1 x2 x1 h5

```

```

rewrite [h6] at h4           --h4 : x1 ∈ X ∧ f x1 = y
show y ∈ image f (W ∩ X) from
  Exists.intro x1 (And.intro (And.intro h3.left h4.left) h3.right)
done
done

```

The rest of Section 5.5 of *HTPI* consists of statements for you to try to prove. Here are the statements, written as examples in Lean. Some are correct and some are not; some can be made correct by adding additional hypotheses or weakening the conclusion. Prove as much as you can.

```

--Warning! Not all of these examples are correct!

example {A B : Type} (f : A → B) (W X : Set A) :
  image f (W ∪ X) = image f W ∪ image f X := sorry

example {A B : Type} (f : A → B) (W X : Set A) :
  image f (W \ X) = image f W \ image f X := sorry

example {A B : Type} (f : A → B) (W X : Set A) :
  W ⊆ X ↔ image f W ⊆ image f X := sorry

example {A B : Type} (f : A → B) (Y Z : Set B) :
  inverse_image f (Y ∩ Z) =
    inverse_image f Y ∩ inverse_image f Z := sorry

example {A B : Type} (f : A → B) (Y Z : Set B) :
  inverse_image f (Y ∪ Z) =
    inverse_image f Y ∪ inverse_image f Z := sorry

example {A B : Type} (f : A → B) (Y Z : Set B) :
  inverse_image f (Y \ Z) =
    inverse_image f Y \ inverse_image f Z := sorry

example {A B : Type} (f : A → B) (Y Z : Set B) :
  Y ⊆ Z ↔ inverse_image f Y ⊆ inverse_image f Z := sorry

example {A B : Type} (f : A → B) (X : Set A) :
  inverse_image f (image f X) = X := sorry

example {A B : Type} (f : A → B) (Y : Set B) :
  image f (inverse_image f Y) = Y := sorry

```

```

example {A : Type} (f : A → A) (C : Set A) :
closed f C → image f C ⊆ C := sorry

example {A : Type} (f : A → A) (C : Set A) :
image f C ⊆ C → C ⊆ inverse_image f C := sorry

example {A : Type} (f : A → A) (C : Set A) :
C ⊆ inverse_image f C → closed f C := sorry

example {A B : Type} (f : A → B) (g : B → A) (Y : Set B)
(h1 : f ∘ g = id) (h2 : g ∘ f = id) :
inverse_image f Y = image g Y := sorry

```

6 Mathematical Induction

6.1. Proof by Mathematical Induction

Section 6.1 of *HTPI* introduces a new proof technique called *mathematical induction*. It is used for proving statements of the form $\forall (n : \text{Nat}), P n$. Here is how it works (*HTPI* p. 273):

To prove a goal of the form $\forall (n : \text{Nat}), P n$:

First prove $P 0$, and then prove $\forall (n : \text{Nat}), P n \rightarrow P (n + 1)$. The first of these proofs is sometimes called the *base case* and the second the *induction step*.

For an explanation of why this strategy works to establish the truth of $\forall (n : \text{Nat}), P n$, see *HTPI*. Here we focus on using mathematical induction in Lean.

To use mathematical induction in a Lean proof, we will use the tactic `by_induc`. If the goal has the form $\forall (n : \text{Nat}), P n$, then the `by_induc` tactic leaves the list of givens unchanged, but it replaces the goal with the goals for the base case and induction step. Thus, the effect of the tactic can be summarized as follows:

Tactic State Before Using Strategy	Tactic State After Using Strategy
<pre>: ↑ ∀ (n : Nat), P n</pre>	<pre>▼ case Base_Case : ↑ P 0 ▼ case Induction_Step : ↑ ∀ (n : Nat), P n → P (n + 1)</pre>

To illustrate proof by mathematical induction in Lean, we turn first to Example 6.1.2 in *HTPI*, which gives a proof of the statement $\forall n \in \mathbb{N} (3 \mid (n^3 - n))$. For reasons that we will explain a little later, we will prove a slightly different theorem: $\forall n \in \mathbb{N} (3 \mid (n^3 + 2n))$. Here is a proof of the theorem, modeled on the proof in Example 6.1.2 of *HTPI* (*HTPI* pp. 276–277).

Theorem. *For every natural number n , $3 \mid (n^3 + 2n)$.*

6.1. Proof by Mathematical Induction

Proof. We use mathematical induction.

Base Case: If $n = 0$, then $n^3 + 2n = 0 = 3 \cdot 0$, so $3 \mid (n^3 + 2n)$.

Induction Step: Let n be an arbitrary natural number and suppose $3 \mid (n^3 + 2n)$. Then we can choose an integer k such that $3k = n^3 + 2n$. Thus,

$$\begin{aligned} (n+1)^3 + 2(n+1) &= n^3 + 3n^2 + 3n + 1 + 2n + 2 \\ &= (n^3 + 2n) + 3n^2 + 3n + 3 \\ &= 3k + 3n^2 + 3n + 3 \\ &= 3(k + n^2 + n + 1). \end{aligned}$$

Therefore $3 \mid ((n+1)^3 + 2(n+1))$, as required. \square

Now let's try writing the same proof in Lean. We start, of course, with the `by_induc` tactic.

Lean File

```
theorem Like_Example_6_1_2 :  
  ∀ (n : Nat), 3 | n ^ 3 + 2 * n := by  
  by_induc  
  done
```

Tactic State in Infoview

```
▼ case Base_Case  
  ⊢ 3 | 0 ^ 3 + 2 * 0  
▼ case Induction_Step  
  ⊢ ∀ (n : ℕ),  
    3 | n ^ 3 + 2 * n →  
    3 | (n + 1) ^ 3 +  
      2 * (n + 1)
```

The base case is easy: The `define` tactic tells us that the goal means $\exists (c : \text{Nat}), 0 ^ 3 + 2 * 0 = 3 * c$, and then `apply Exists.intro 0` changes the goal to $0 ^ 3 + 2 * 0 = 3 * 0$. Both sides are definitionally equal to 0, so `rfl` finishes off the base case. For the induction step, we begin, as in the *HTPI* proof, by introducing an arbitrary natural number n and assuming $3 \mid n ^ 3 + 2 * n$. This assumption is called the *inductive hypothesis*, so in the Lean proof we give it the identifier `ih`. Our goal now is to prove $3 \mid (n + 1) ^ 3 + 2 * (n + 1)$.

6.1. Proof by Mathematical Induction

Lean File

```
theorem Like_Example_6_1_2 :  
  ∀ (n : Nat), 3 | n ^ 3 + 2 * n := by  
by_induc  
• -- Base Case  
define  
apply Exists.intro 0  
rfl  
done  
• -- Induction Step  
fix n : Nat  
assume ih : 3 | n ^ 3 + 2 * n  
done  
done
```

Tactic State in Infoview

```
▼case Induction_Step  
n : ℕ  
ih : 3 | n ^ 3 + 2 * n  
↑ 3 | (n + 1) ^ 3 +  
2 * (n + 1)
```

The rest of the Lean proof follows the model of the *HTPI* proof: we use the inductive hypothesis to introduce a k such that $n^3 + 2 \cdot n = 3 \cdot k$, and then we use a calculational proof to show that $(n + 1)^3 + 2 \cdot (n + 1) = 3 \cdot (k + n^2 + n + 1)$.

```
theorem Like_Example_6_1_2 :  
  ∀ (n : Nat), 3 | n ^ 3 + 2 * n := by  
by_induc  
• -- Base Case  
define      --Goal : ∃ (c : Nat), 0 ^ 3 + 2 * 0 = 3 * c  
apply Exists.intro 0  
rfl  
done  
• -- Induction Step  
fix n : Nat  
assume ih : 3 | n ^ 3 + 2 * n  
define at ih  --ih : ∃ (c : Nat), n ^ 3 + 2 * n = 3 * c  
obtain (k : Nat) (h1 : n ^ 3 + 2 * n = 3 * k) from ih  
define      --Goal : ∃ (c : Nat), (n + 1) ^ 3 + 2 * (n + 1) = 3 * c  
apply Exists.intro (k + n ^ 2 + n + 1)  
show (n + 1) ^ 3 + 2 * (n + 1) = 3 * (k + n ^ 2 + n + 1) from  
calc (n + 1) ^ 3 + 2 * (n + 1)  
_ = n ^ 3 + 2 * n + 3 * n ^ 2 + 3 * n + 3 := by ring  
_ = 3 * k + 3 * n ^ 2 + 3 * n + 3 := by rw [h1]  
_ = 3 * (k + n ^ 2 + n + 1) := by ring  
done  
done
```

Next we'll look at Example 6.1.1 in *HTPI*, which proves that for every natural number n ,

6.1. Proof by Mathematical Induction

$2^0 + 2^1 + \cdots + 2^n = 2^{n+1} - 1$. Once again, we will change the theorem slightly before proving it in Lean. What we will prove is that for every n , $(2^0 + 2^1 + \cdots + 2^n) + 1 = 2^{n+1}$. To understand this theorem you must be able to recognize what the “...” stands for. A human reader will probably realize that the numbers being added up here are the numbers of the form 2^i , where i runs through all of the natural numbers from 0 to n . But Lean can't be expected to figure out this pattern, so we must be more explicit.

Section 6.3 of *HTPI* introduces the explicit notation that mathematicians usually use for such sums. If f is a function whose domain is the natural numbers, then

$$\sum_{i=0}^n f(i) = f(0) + f(1) + \cdots + f(n).$$

More generally, if $k \leq n$ then

$$\sum_{i=k}^n f(i) = f(k) + f(k+1) + \cdots + f(n).$$

The notation we will use in Lean for this sum is `Sum i from k to n, f i`. Thus, a mathematician would state our theorem like this:

Theorem. *For every natural number n ,*

$$\left(\sum_{i=0}^n 2^i \right) + 1 = 2^{n+1}.$$

And to state the same theorem in Lean, we will write:

```
theorem Like_Example_6_1_1 :  
  ∀ (n : Nat), (Sum i from 0 to n, 2 ^ i) + 1 = 2 ^ (n + 1)
```

We will have more to say later about how the notation `Sum i from k to n, f i` is defined. But to use the notation in a proof, we will just need to know a few theorems. The `#check` command will tell us the meanings of the theorems `sum_base`, `sum_step`, and `sum_from_zero_step`:

```
@sum_base : ∀ {A : Type} [inst : AddZeroClass A] {k : ℕ} {f : ℕ → A},  
  Sum i from k to k, f i = f k  
  
@sum_step : ∀ {A : Type} [inst : AddZeroClass A] {k n : ℕ} {f : ℕ → A},  
  k ≤ n → Sum i from k to n + 1, f i =  
    (Sum i from k to n, f i) + f (n + 1)  
  
@sum_from_zero_step :
```

6.1. Proof by Mathematical Induction

```

$$\forall \{A : \text{Type}\} [\text{inst} : \text{AddZeroClass } A] \{n : \mathbb{N}\} \{f : \mathbb{N} \rightarrow A\},$$


$$\text{Sum } i \text{ from } 0 \text{ to } n + 1, f i =$$


$$(\text{Sum } i \text{ from } 0 \text{ to } n, f i) + f (n + 1)$$

```

As usual, we don't need to pay too much attention to the implicit arguments in the first line of each statement. What is important is that `sum_base` can be used to prove any statement of the form

```
Sum i from k to k, f i = f k
```

and `sum_step` proves any statement of the form

```
 $k \leq n \rightarrow \text{Sum } i \text{ from } k \text{ to } n + 1, f i = (\text{Sum } i \text{ from } k \text{ to } n, f i) + f (n + 1).$ 
```

In the case $k = 0$, we have the simpler theorem `sum_from_zero_step`, which proves

```
Sum i from 0 to n + 1, f i = (Sum i from 0 to n, f i) + f (n + 1).
```

With that preparation, we can start on the proof. Once again we begin with the `by_induc` tactic. Our goal for the base case is $(\text{Sum } i \text{ from } 0 \text{ to } 0, 2^i) + 1 = 2^{(0+1)}$. To deal with the term $\text{Sum } i \text{ from } 0 \text{ to } 0, 2^i$, we use that fact that `sum_base` proves $\text{Sum } i \text{ from } 0 \text{ to } 0, 2^i = 2^0$. It follows that the tactic `rewrite [sum_base]` will change the goal to $2^0 + 1 = 2^{(0+1)}$. Of course, this means $2 = 2$, so `rfl` finishes the base case. For the induction step, we start by introducing an arbitrary natural number n and assuming the inductive hypothesis.

```
theorem Like_Example_6_1_1 :
  ∀ (n : Nat), (Sum i from 0 to n, 2^i) + 1 = 2^(n + 1) := by
  by_induc
  · -- Base Case
    rewrite [sum_base]
    rfl
    done
  · -- Induction Step
    fix n : Nat
    assume ih : (Sum i from 0 to n, 2^i) + 1 = 2^(n + 1)
    done
    done
```

Our goal is now $(\text{Sum } i \text{ from } 0 \text{ to } n + 1, 2^i) + 1 = 2^{(n+1+1)}$, and we use a calculational proof to prove this. Often the key to the proof of the induction step is to find a relationship between the inductive hypothesis and the goal. In this case, that means finding a relationship between $\text{Sum } i \text{ from } 0 \text{ to } n, 2^i$ and $\text{Sum } i \text{ from } 0 \text{ to } n + 1, 2^i$. The relationship we need is given by the theorem `sum_from_zero_step`. The tactic `rewrite [sum_from_zero_step]`

6.1. Proof by Mathematical Induction

will replace `Sum i from 0 to n + 1, 2 ^ i` with `(Sum i from 0 to n, 2 ^ i) + 2 ^ (n + 1)`. The rest of the calculation proof involves straightforward algebra, handled by the `ring` tactic, together with an application of the inductive hypothesis.

```
theorem Like_Example_6_1_1 : 
  ∀ (n : Nat), (Sum i from 0 to n, 2 ^ i) + 1 = 2 ^ (n + 1) := by
  by_induc
  . -- Base Case
    rewrite [sum_base]
    rfl
    done
  . -- Induction Step
    fix n : Nat
    assume ih : (Sum i from 0 to n, 2 ^ i) + 1 = 2 ^ (n + 1)
    show (Sum i from 0 to n + 1, 2 ^ i) + 1 = 2 ^ (n + 1 + 1) from
      calc (Sum i from 0 to n + 1, 2 ^ i) + 1
        _ = (Sum i from 0 to n, 2 ^ i) + 2 ^ (n + 1) + 1 := by
          rw [sum_from_zero_step]
        _ = (Sum i from 0 to n, 2 ^ i) + 1 + 2 ^ (n + 1) := by ring
        _ = 2 ^ (n + 1) + 2 ^ (n + 1) := by rw [ih]
        _ = 2 ^ (n + 1 + 1) := by ring
    done
done
```

The last example in Section 6.1 of *HTPI* gives a proof of the statement $\forall n \geq 5 (2^n > n^2)$. The proof is by mathematical induction, but since we are only interested in natural numbers greater than or equal to 5, it uses 5 in the base case instead of 0. Here are the theorem and proof from *HTPI* (*HTPI* p. 278):

Theorem. *For every natural number $n \geq 5$, $2^n > n^2$.*

Proof. By mathematical induction.

Base case: When $n = 5$ we have $2^n = 32 > 25 = n^2$.

Induction step: Let $n \geq 5$ be arbitrary, and suppose that $2^n > n^2$. Then

$$\begin{aligned} 2^{n+1} &= 2 \cdot 2^n \\ &> 2n^2 && \text{(inductive hypothesis)} \\ &= n^2 + n^2 \\ &\geq n^2 + 5n && \text{(since } n \geq 5\text{)} \\ &= n^2 + 2n + 3n \\ &> n^2 + 2n + 1 = (n + 1)^2. \end{aligned}$$

□

6.1. Proof by Mathematical Induction

Notice that the sequence of calculations at the end of the proof mixes $=$, $>$, and \geq in a way that establishes the final conclusion $2^{n+1} > (n + 1)^2$. As we'll see, such a mixture is allowed in calculational proofs in Lean as well.

To write this proof in Lean, there is no need to specify that the base case should be $n = 5$; the `by_induc` tactic is smart enough to figure that out on its own, as you can see in the tactic state below. (Notice that, as in *HTPI*, in Lean we can write $\forall n \geq k, P n$ as an abbreviation for $\forall (n : \text{Nat}), n \geq k \rightarrow P n$.)

Lean File

```
theorem Example_6_1_3 :  
  ∀ n ≥ 5, 2 ^ n > n ^ 2 := by  
  by_induc  
  done
```

Tactic State in Infoview

```
▼ case Base_Case  
  ⊢ 2 ^ 5 > 5 ^ 2  
▼ case Induction_Step  
  ⊢ ∀ n ≥ 5,  
    2 ^ n > n ^ 2 →  
    2 ^ (n + 1) >  
    (n + 1) ^ 2
```

To complete this proof we'll use two tactics we haven't used before: `decide` and `linarith`. The truth or falsity of the inequality in the base case can be decided by simply doing the necessary arithmetic. The tactic `decide` can do such calculations, and it proves the base case.

For the induction step, we introduce an arbitrary natural number n , assume $n \geq 5$, and assume the inductive hypothesis, $2 ^ n > n ^ 2$. Then we use a calculational proof to imitate the reasoning at the end of the *HTPI* proof. The tactic `linarith` makes inferences that involve combining linear equations and inequalities. It is able to prove almost all of the inequalities in the calculational proof. The exception is $n * n \geq 5 * n$ (which is not linear because of the term $n * n$). So we prove that inequality separately, using a theorem from Lean's library, `Nat.mul_le_mul_right`. The command `#check @Nat.mul_le_mul_right` tells us the meaning of that theorem:

```
@Nat.mul_le_mul_right : ∀ {n m : ℕ} (k : ℕ), n ≤ m → n * k ≤ m * k
```

Thus, `Nat.mul_le_mul_right n` can be used to prove the statement $5 \leq n \rightarrow 5 * n \leq n * n$. Lean recognizes $x \geq y$ as meaning the same thing as $y \leq x$, so we can apply this statement to our assumption $n \geq 5$ to prove that $n * n \geq 5 * n$. Once we have proven that inequality, the `linarith` tactic can use it to complete the required inequality reasoning.

```
theorem Example_6_1_3 : ∀ n ≥ 5, 2 ^ n > n ^ 2 := by  
  by_induc  
  . -- Base Case  
    decide  
    done  
  . -- Induction Step
```

```

fix n : Nat
assume h1 : n ≥ 5
assume ih : 2 ^ n > n ^ 2
have h2 : n * n ≥ 5 * n := Nat.mul_le_mul_right n h1
show 2 ^ (n + 1) > (n + 1) ^ 2 from
calc 2 ^ (n + 1)
_ = 2 * 2 ^ n := by ring
_ > 2 * n ^ 2 := by linarith
_ ≥ n ^ 2 + 5 * n := by linarith
_ > n ^ 2 + 2 * n + 1 := by linarith
_ = (n + 1) ^ 2 := by ring
done
done

```

Finally, we turn to the question of why we made small changes in two of the examples from *HTPI*. Perhaps you have guessed by now that we were trying to avoid the use of subtraction. All of the numbers in the examples in this section were natural numbers, and subtraction of natural numbers is problematic. In the natural numbers, $3 - 2$ is equal to 1, but what is $2 - 3$? Lean's answer is 0.

```
#eval 2 - 3      --Answer: 0
```

In Lean, if a and b are natural numbers and $a < b$, then $a - b$ is defined to be 0. As a result, the algebraic laws of natural number subtraction are complicated. For example, $2 - 3 + 1 = 0 + 1 = 1$, but $2 + 1 - 3 = 3 - 3 = 0$, so it is not true that for all natural numbers a , b , and c , $a - b + c = a + c - b$.

If you thought that the answer to the subtraction problem $2 - 3$ was -1 , then you automatically switched from the natural numbers to the integers. (Recall that the natural numbers are the numbers $0, 1, 2, \dots$, while the integers are the numbers $\dots, -3, -2, -1, 0, 1, 2, 3, \dots$) To a human mathematician, this is a perfectly natural thing to do: the natural numbers are a subset of the integers, so 2 and 3 are not only natural numbers but also integers, and we can compute $2 - 3$ in the integers.

However, that's not how things work in Lean. In Lean, different types are completely separate. In particular, `Nat` and `Int` are separate types, and therefore the natural numbers are not a subset of the integers. Of course, there is an integer 2, but it is different from the natural number 2. By default, Lean assumes that 2 denotes the natural number 2, but you can specify that you want the integer 2 by writing `(2 : Int)`. Subtraction of integers in Lean is the subtraction you are familiar with, and it has all the algebraic properties you would expect. If we want to use subtraction in the theorems in this section, we are better off using familiar integer subtraction rather than funky natural number subtraction.

6.1. Proof by Mathematical Induction

To prove the theorem in Example 6.1.1 as it appears in *HTPI*, we could state the theorem like this:

```
theorem Example_6_1_1 :  
  ∀ (n : Nat), Sum i from 0 to n, (2 : Int) ^ i =  
    (2 : Int) ^ (n + 1) - (1 : Int)
```

The expression `Sum i from 0 to n, (2 : Int) ^ i` denotes a sum of integers, so it is an integer. Similarly, the right side of the equation is an integer, and the equation asserts the equality of two integers. The subtraction on the right side of the equation is integer subtraction, so we can use the usual algebraic laws to reason about it. In fact, the proof of the theorem in this form is not hard:

```
theorem Example_6_1_1 :  
  ∀ (n : Nat), Sum i from 0 to n, (2 : Int) ^ i =  
    (2 : Int) ^ (n + 1) - (1 : Int) := by  
  by_induc  
  · -- Base Case  
    rewrite [sum_base]  
    rfl  
    done  
  · -- Induction Step  
    fix n : Nat  
    assume ih : Sum i from 0 to n, (2 : Int) ^ i =  
      (2 : Int) ^ (n + 1) - (1 : Int)  
    show Sum i from 0 to n + 1, (2 : Int) ^ i =  
      (2 : Int) ^ (n + 1 + 1) - (1 : Int) from  
    calc Sum i from 0 to n + 1, (2 : Int) ^ i  
    _ = (Sum i from 0 to n, (2 : Int) ^ i)  
      + (2 : Int) ^ (n + 1) := by rw [sum_from_zero_step]  
    _ = (2 : Int) ^ (n + 1) - (1 : Int)  
      + (2 : Int) ^ (n + 1) := by rw [ih]  
    _ = (2 : Int) ^ (n + 1 + 1) - (1 : Int) := by ring  
    done  
  done
```

If you change `(2 : Int)` and `(1 : Int)` to `2` and `1`, then the right side of the equation will be a difference of two natural numbers, and Lean will interpret the subtraction as natural number subtraction. The proof won't work because the `ring` tactic is not able to deal with the peculiar algebraic properties of natural number subtraction. (The theorem is still true, but the proof is harder.)

Exercises

1. `theorem Like_Exercise_6_1_1 :
 ∀ (n : Nat), 2 * Sum i from 0 to n, i = n * (n + 1) := sorry`
2. `theorem Like_Exercise_6_1_4 :
 ∀ (n : Nat), Sum i from 0 to n, 2 * i + 1 = (n + 1) ^ 2 := sorry`
3. `theorem Exercise_6_1_9a : ∀ (n : Nat), 2 | n ^ 2 + n := sorry`
4. `theorem Exercise_6_1_13 :
 ∀ (a b : Int) (n : Nat), (a - b) | (a ^ n - b ^ n) := sorry`
5. `theorem Exercise_6_1_15 : ∀ n ≥ 10, 2 ^ n > n ^ 3 := sorry`
6. `lemma nonzero_is_successor :
 ∀ (n : Nat), n ≠ 0 → ∃ (m : Nat), n = m + 1 := sorry`

For the next two exercises you will need the following definitions:

```
def nat_even (n : Nat) : Prop := ∃ (k : Nat), n = 2 * k

def nat_odd (n : Nat) : Prop := ∃ (k : Nat), n = 2 * k + 1
```

7. `theorem Exercise_6_1_16a1 :
 ∀ (n : Nat), nat_even n ∨ nat_odd n := sorry`
8. `--Hint: You may find the lemma nonzero_is_successor
--from a previous exercise useful, as well as Nat.add_right_cancel.
theorem Exercise_6_1_16a2 :
 ∀ (n : Nat), ¬(nat_even n ∧ nat_odd n) := sorry`

6.2. More Examples

We saw in the last section that mathematical induction can be used to prove theorems about calculations involving natural numbers. But mathematical induction has a much wider range of uses. Section 6.2 of *HTPI* illustrates this by proving two theorems about finite sets.

How can mathematical induction be used to prove a statement about finite sets? To say that a set is finite means that it has n elements, for some natural number n . Thus, to say that all finite sets have some property, we can say that for every natural number n , every set with n elements has the property. Since this statement starts with “for every natural number n ,” we can use mathematical induction to try to prove it.

6.2. More Examples

What does it mean to say that a set “has n elements”? Section 6.2 of *HTPI* says that for the proofs in that section, “an intuitive understanding of this concept will suffice.” Unfortunately, intuition is not Lean’s strong suit! So we’ll need to be more explicit about how to talk about finite sets in Lean.

In Chapter 8, we’ll define `numElts A n` to be a proposition saying that the set A has n elements, and we’ll prove several theorems involving that proposition. Those theorems make precise and explicit the intuitive ideas that we’ll need in this section. We’ll state those theorems here and use them in our proofs, but you’ll have to wait until Section 8.1½ to see how they are proven. Here are the theorems we’ll need:

```
theorem zero_elts_iff_empty {U : Type} (A : Set U) :
  numElts A 0 ↔ empty A

theorem one_elt_iff_singleton {U : Type} (A : Set U) :
  numElts A 1 ↔ ∃ (x : U), A = {x}

theorem nonempty_of_pos_numElts {U : Type} {A : Set U} {n : Nat}
  (h1 : numElts A n) (h2 : n > 0) : ∃ (x : U), x ∈ A

theorem remove_one_numElts {U : Type} {A : Set U} {n : Nat} {a : U}
  (h1 : numElts A (n + 1)) (h2 : a ∈ A) : numElts (A \ {a}) n
```

These theorems should make intuitive sense. The first says that a set has zero elements if and only if it is empty, and the second says that a set has one element if and only if it is a singleton set. The third theorem says that if a set has a positive number of elements, then there is something in the set. And the fourth says that if a set has $n + 1$ elements and you remove one element, then the resulting set has n elements. You can probably guess that we’ll be using the last theorem in the induction steps of our proofs.

Our first theorem about finite sets says that if R is a partial order on A , then every finite, nonempty subset of A has an R -minimal element. (This is not true in general for infinite subsets of A . Can you think of an example of an infinite subset of a partially ordered set that has no minimal element?) To say that a set is finite and nonempty we can say that it has n elements for some $n \geq 1$. So here’s how we state our theorem in Lean:

```
theorem Example_6_2_1 {A : Type} (R : BinRel A) (h : partial_order R) :
  ∀ n ≥ 1, ∀ (B : Set A), numElts B n →
    ∃ (x : A), minimalElt R x B
```

When we use mathematical induction to prove this theorem, the base case will be $n = 1$. To write the proof for the base case, we start by assuming B is a set with one element. We can then use the theorem `one_elt_iff_singleton` to conclude that $B = \{b\}$, for some b of type A . We

6.2. More Examples

need to prove that B has a minimal element, and the only possibility for the minimal element is b . Verifying that `minimalElt R b B` is straightforward. Here is the proof of the base case:

```
theorem Example_6_2_1 {A : Type} (R : BinRel A) (h : partial_order R) :
  ∀ n ≥ 1, ∀ (B : Set A), numElts B n →
    ∃ (x : A), minimalElt R x B := by
  by_induc
  . -- Base Case
  fix B : Set A
  assume h2 : numElts B 1
  rewrite [one_elt_iff_singleton] at h2
  obtain (b : A) (h3 : B = {b}) from h2
  apply Exists.intro b
  define      --Goal : b ∈ B ∧ ¬∃ x ∈ B, R x b ∧ x ≠ b
  apply And.intro
  . -- Proof that b ∈ B
  rewrite [h3]      --Goal : b ∈ {b}
  define      --Goal : b = b
  rfl
  done
  . -- Proof that nothing in B is smaller than b
  by_contra h4
  obtain (x : A) (h5 : x ∈ B ∧ R x b ∧ x ≠ b) from h4
  have h6 : x ∈ B := h5.left
  rewrite [h3] at h6      --h6 : x ∈ {b}
  define at h6      --h6 : x = b
  show False from h5.right.right h6
  done
  done
  . -- Induction Step

  done
done
```

Notice that since the definition of `minimalElt R b B` involves a negative statement, we found it convenient to use proof by contradiction to prove it.

For the induction step, we assume that $n \geq 1$ and that every set with n elements has an R -minimal element. We must prove that every set with $n + 1$ elements has a minimal element, so we let B be an arbitrary set with $n + 1$ elements. To apply the inductive hypothesis, we need a set with n elements. So we pick some $b \in B$ (using the theorem `nonempty_of_pos_numElts`) and then remove it from B to get the set $B' = B \setminus \{b\}$. The theorem `remove_one_numElts` tells us that B' has n elements, so by the inductive hypothesis, we can then let c be a minimal element

6.2. More Examples

of B' . We now know about two elements of B : b and c . Which will be a minimal element of B ? As explained in *HTPI*, it depends on whether or not $R b c$. We'll prove that if $R b c$, then b is a minimal element of B , and if not, then c is a minimal element. It will be convenient to prove these last two facts separately as lemmas. The first lemma says that in the situation at this point in the proof, if $R b c$, then b is an R -minimal element of B . Here is the proof.

```
lemma Lemma_6_2_1_1 {A : Type} {R : BinRel A} {B : Set A} {b c : A}
  (h1 : partial_order R) (h2 : b ∈ B) (h3 : minimalElt R c (B \ {b}))
  (h4 : R b c) : minimalElt R b B := by
  define at h3
    --h3 : c ∈ B \ {b} ∧ ¬∃ x ∈ B \ {b}, R x c ∧ x ≠ c
  define --Goal : b ∈ B ∧ ¬∃ x ∈ B, R x b ∧ x ≠ b
  apply And.intro h2    --Goal : ¬∃ x ∈ B, R x b ∧ x ≠ b
  contradict h3.right with h5
  obtain (x : A) (h6 : x ∈ B ∧ R x b ∧ x ≠ b) from h5
  apply Exists.intro x --Goal : x ∈ B \ {b} ∧ R x c ∧ x ≠ c
  apply And.intro
  • -- Proof that x ∈ B \ {b}
    show x ∈ B \ {b} from And.intro h6.left h6.right.right
    done
  • -- Proof that R x c ∧ x ≠ c
    have Rtrans : transitive R := h1.right.left
    have h7 : R x c := Rtrans x b c h6.right.left h4
    apply And.intro h7
    by_contra h8
    rewrite [h8] at h6 --h6 : c ∈ B ∧ R c b ∧ c ≠ b
    have Rantisymm : antisymmetric R := h1.right.right
    have h9 : c = b := Rantisymm c b h6.right.left h4
    show False from h6.right.right h9
    done
  done
```

The second lemma says that if $\neg R b c$, then c is an R -minimal element of B . We'll leave the proof as an exercise for you:

```
lemma Lemma_6_2_1_2 {A : Type} {R : BinRel A} {B : Set A} {b c : A}
  (h1 : partial_order R) (h2 : b ∈ B) (h3 : minimalElt R c (B \ {b}))
  (h4 : ¬R b c) : minimalElt R c B := sorry
```

With this preparation, we are finally ready to give the proof of the induction step of Example_6_2_1:

6.2. More Examples

```

theorem Example_6_2_1 {A : Type} (R : BinRel A) (h : partial_order R) :
  ∀ n ≥ 1, ∀ (B : Set A), numElts B n →
    ∃ (x : A), minimalElt R x B := by
  by_induc
  · -- Base Case
  ...
  · -- Induction Step
  fix n : Nat
  assume h2 : n ≥ 1
  assume ih : ∀ (B : Set A), numElts B n → ∃ (x : A), minimalElt R x B
  fix B : Set A
  assume h3 : numElts B (n + 1)
  have h4 : n + 1 > 0 := by linarith
  obtain (b : A) (h5 : b ∈ B) from nonempty_of_pos_numElts h3 h4
  set B' : Set A := B \ {b}
  have h6 : numElts B' n := remove_one_numElts h3 h5
  obtain (c : A) (h7 : minimalElt R c B') from ih B' h6
  by_cases h8 : R b c
  · -- Case 1. h8 : R b c
    have h9 : minimalElt R b B := Lemma_6_2_1_1 h h5 h7 h8
    show ∃ (x : A), minimalElt R x B from Exists.intro b h9
    done
  · -- Case 2. h8 : ¬R b c
    have h9 : minimalElt R c B := Lemma_6_2_1_2 h h5 h7 h8
    show ∃ (x : A), minimalElt R x B from Exists.intro c h9
    done
  done
done

```

We'll consider one more theorem from Section 6.2 of *HTPI*. Example 6.2.2 proves that a partial order on a finite set can always be extended to a total order. Rather than give that proof, we are going to prove the more general theorem that is stated in Exercise 2 in Section 6.2 of *HTPI*. To explain the theorem in that exercise, it will be helpful to introduce a bit of terminology. Suppose R is a partial order on A and b has type A . We will say that b is *R -comparable to everything* if $\forall (x : A), R b x \vee R x b$. If B is a set of objects of type A , we say that B is *R -comparable to everything* if every element of B is R -comparable to everything; that is, if $\forall b \in B, \forall (x : A), R b x \vee R x b$. Finally, we say that another binary relation T *extends* R if $\forall (x y : A), R x y \rightarrow T x y$. We are going to prove that if R is a partial order on A and B is a finite set of objects of type A , then there is a partial order T that extends R such that B is T -comparable to everything. In other words, we are going to prove the following theorem:

6.2. More Examples

```
theorem Exercise_6_2_2 {A : Type} (R : BinRel A) (h : partial_order R) :
  ∀ (n : Nat) (B : Set A), numElts B n → ∃ (T : BinRel A),
  partial_order T ∧ (∀ (x y : A), R x y → T x y) ∧
  ∀ x ∈ B, ∀ (y : A), T x y ∨ T y x
```

In the exercises, we will ask you to show that this implies the theorem in Example 6.2.2.

It will be helpful to begin with a warm-up exercise. We'll show that a partial order can always be extended to make a single object comparable to everything. In other words, we'll show that if R is a partial order on A and b has type A , then we can define a partial order T extending R such that b is T -comparable to everything. To define T , we will need to make sure that for every x of type A , either $T b x$ or $T x b$. If $R x b$, then since T must extend R , we must have $T x b$. If $\neg R x b$, then we will define T so that $T b x$. But notice that if we follow this plan, then for any x and y , if we have $R x b$ and $\neg R y b$, then we will have $T x b$ and $T b y$, and since T must be transitive, we must then have $T x y$. Summing up, if we have $R x y$ then we must have $T x y$, and if we have $R x b$ and $\neg R y b$ then we will also need to have $T x y$. So let's try defining $T x y$ to mean $R x y \vee (R x b \wedge \neg R y b)$.

It will be useful to have a name for this relation T . Since it is an extension of R determined by the element b , we will give it the name `extendPO R b`. Here is the definition of this relation:

```
def extendPO {A : Type} (R : BinRel A) (b : A) (x y : A) : Prop :=
  R x y ∨ (R x b ∧ ¬R y b)
```

We need to prove a number of things about `extendPO R b`. First of all, we need to prove that it is a partial order. We'll leave most of the details as exercises for you:

```
lemma extendPO_is_ref {A : Type} (R : BinRel A) (b : A)
  (h : partial_order R) : reflexive (extendPO R b) := sorry

lemma extendPO_is_trans {A : Type} (R : BinRel A) (b : A)
  (h : partial_order R) : transitive (extendPO R b) := sorry

lemma extendPO_is_antisymm {A : Type} (R : BinRel A) (b : A)
  (h : partial_order R) : antisymmetric (extendPO R b) := sorry

lemma extendPO_is_po {A : Type} (R : BinRel A) (b : A)
  (h : partial_order R) : partial_order (extendPO R b) :=
  And.intro (extendPO_is_ref R b h)
  (And.intro (extendPO_is_trans R b h) (extendPO_is_antisymm R b h))
```

It is easy to prove that `extendPO R b` extends R :

6.2. More Examples

```
lemma extendPO_extends {A : Type} (R : BinRel A) (b : A) (x y : A) :
  R x y → extendPO R b x y := by
  assume h1 : R x y
  define
  show R x y ∨ R x b ∧ ¬R y b from Or.inl h1
  done
```

Finally, we verify that `extendPO R b` does what it was supposed to do: it makes `b` comparable with everything:

```
lemma extendPO_all_comp {A : Type} (R : BinRel A) (b : A)
  (h : partial_order R) :
  ∀ (x : A), extendPO R b b x ∨ extendPO R b x b := by
  have Rref : reflexive R := h.left
  fix x : A
  or_left with h1
  define at h1    --h1 : ¬(R x b ∨ R x b ∧ ¬R b b)
  demorgan at h1 --h1 : ¬R x b ∧ ¬(R x b ∧ ¬R b b)
  define         --Goal : R b x ∨ R b b ∧ ¬R x b
  apply Or.inr
  show R b b ∧ ¬R x b from And.intro (Rref b) h1.left
  done
```

With this preparation, we can finally return to our theorem `Exercise_6_2_2`. We will prove it by mathematical induction. In the base case we must show that if B has 0 elements then we can extend R to make everything in B comparable to everything. Of course, no extension is necessary, since it is vacuously true that all elements of B are R -comparable to everything. For the induction step, after assuming the inductive hypothesis, we must prove that if B has $n + 1$ elements then we can extend R to make all elements of B comparable to everything. As before, we choose $b \in B$ and let $B' = B \setminus \{b\}$. By inductive hypothesis, we can find an extension T' of R that makes all elements of B' comparable to everything, so we just have to extend T' further to make b comparable to everything. But as we have just seen, we can do this with `extendPO T' b`.

```
theorem Exercise_6_2_2 {A : Type} (R : BinRel A) (h : partial_order R) :
  ∀ (n : Nat) (B : Set A), numElts B n → ∃ (T : BinRel A),
  partial_order T ∧ (∀ (x y : A), R x y → T x y) ∧
  ∀ x ∈ B, ∀ (y : A), T x y ∨ T y x := by
  by_induc
  · -- Base Case
  fix B : Set A
  assume h2 : numElts B 0
```

```

rewrite [zero_elts_iff_empty] at h2
define at h2      --h2 :  $\neg\exists (x : A), x \in B$ 
apply Exists.intro R
apply And.intro h
apply And.intro
• -- Proof that R extends R
  fix x : A; fix y : A
  assume h3 : R x y
  show R x y from h3
  done
• -- Proof that everything in B comparable to everything under R
  fix x : A
  assume h3 : x ∈ B
  contradict h2
  show  $\exists (x : A), x \in B$  from Exists.intro x h3
  done
done
• -- Induction Step
fix n : Nat
assume ih :  $\forall (B : \text{Set } A), \text{numElts } B n \rightarrow \exists (T : \text{BinRel } A),$ 
  partial_order T  $\wedge (\forall (x y : A), R x y \rightarrow T x y) \wedge$ 
   $\forall (x : A), x \in B \rightarrow \forall (y : A), T x y \vee T y x$ 
fix B : Set A
assume h2 : numElts B (n + 1)
have h3 : n + 1 > 0 := by linarith
obtain (b : A) (h4 : b ∈ B) from nonempty_of_pos_numElts h2 h3
set B' : Set A := B \ {b}
have h5 : numElts B' n := remove_one_numElts h2 h4
have h6 :  $\exists (T : \text{BinRel } A), \text{partial\_order } T \wedge$ 
   $(\forall (x y : A), R x y \rightarrow T x y) \wedge$ 
   $\forall (x : A), x \in B' \rightarrow \forall (y : A), T x y \vee T y x := ih B' h5$ 
obtain (T' : BinRel A)
  (h7 : partial_order T'  $\wedge (\forall (x y : A), R x y \rightarrow T' x y) \wedge$ 
    $\forall (x : A), x \in B' \rightarrow \forall (y : A), T' x y \vee T' y x)$  from h6
have T'po : partial_order T' := h7.left
have T'extR :  $\forall (x y : A), R x y \rightarrow T' x y := h7.right.left$ 
have T'compB' :  $\forall (x : A), x \in B' \rightarrow$ 
   $\forall (y : A), T' x y \vee T' y x := h7.right.right$ 
set T : BinRel A := extendPO T' b
apply Exists.intro T
apply And.intro (extendPO_is_po T' b T'po)
apply And.intro

```

```

• -- Proof that T extends R
  fix x : A; fix y : A
  assume h8 : R x y
  have h9 : T' x y := T'extR x y h8
  show T x y from (extendPO_extends T' b x y h9)
  done
• -- Proof that everything in B comparable to everything under T
  fix x : A
  assume h8 : x ∈ B
  by_cases h9 : x = b
  • -- Case 1. h9 : x = b
    rewrite [h9]
    show ∀ (y : A), T b y ∨ T y b from extendPO_all_comp T' b T'po
    done
  • -- Case 2. h9 : x ≠ b
    have h10 : x ∈ B' := And.intro h8 h9
    fix y : A
    have h11 : T' x y ∨ T' y x := T'compB' x h10 y
    by_cases on h11
    • -- Case 2.1. h11 : T' x y
      show T x y ∨ T y x from
        Or.inl (extendPO_extends T' b x y h11)
      done
    • -- Case 2.2. h11 : T' y x
      show T x y ∨ T y x from
        Or.inr (extendPO_extends T' b y x h11)
      done
    done
  done
done
done
done

```

Exercises

1. `lemma Lemma_6_2_1_2 {A : Type} {R : BinRel A} {B : Set A} {b c : A}`
`(h1 : partial_order R) (h2 : b ∈ B) (h3 : minimalElt R c (B \ {b}))`
`(h4 : ¬R b c) : minimalElt R c B := sorry`
2. `lemma extendPO_is_ref {A : Type} (R : BinRel A) (b : A)`
`(h : partial_order R) : reflexive (extendPO R b) := sorry`

```

3. lemma extendPO_is_trans {A : Type} (R : BinRel A) (b : A)
   (h : partial_order R) : transitive (extendPO R b) := sorry

4. lemma extendPO_is_antisymm {A : Type} (R : BinRel A) (b : A)
   (h : partial_order R) : antisymmetric (extendPO R b) := sorry

5. theorem Exercise_6_2_3 {A : Type} (R : BinRel A)
   (h : total_order R) : ∀ n ≥ 1, ∀ (B : Set A),
   numElts B n → ∃ (b : A), smallestElt R b B := sorry

6. --Hint: First prove that R is reflexive.
theorem Exercise_6_2_4a {A : Type} (R : BinRel A)
  (h : ∀ (x y : A), R x y ∨ R y x) : ∀ n ≥ 1, ∀ (B : Set A),
  numElts B n → ∃ x ∈ B, ∀ y ∈ B, ∃ (z : A), R x z ∧ R z y := sorry

7. theorem Like_Exercise_6_2_16 {A : Type} (f : A → A)
  (h : one_to_one f) : ∀ (n : Nat) (B : Set A), numElts B n →
  closed f B → ∀ y ∈ B, ∃ x ∈ B, f x = y := sorry

8. --Hint: Use Exercise_6_2_2.
theorem Example_6_2_2 {A : Type} (R : BinRel A)
  (h1 : ∃ (n : Nat), numElts {x : A | x = x} n)
  (h2 : partial_order R) : ∃ (T : BinRel A),
  total_order T ∧ ∀ (x y : A), R x y → T x y := sorry

```

6.3. Recursion

In the last two sections, we saw that we can prove that all natural numbers have some property by proving that 0 has the property, and also that for every natural number n , if n has the property then so does $n+1$. In this section we will see that a similar idea can be used to define a function whose domain is the natural numbers. We can define a function f with domain \mathbb{N} by specifying the value of $f(0)$, and also saying how to compute $f(n+1)$ if you already know the value of $f(n)$.

For example, we can define a function $f : \mathbb{N} \rightarrow \mathbb{N}$ as follows:

$$\begin{aligned} f(0) &= 1; \\ \text{for every } n \in \mathbb{N}, f(n+1) &= (n+1) \cdot f(n). \end{aligned}$$

Here is the same definition written in Lean. (For reasons that will become clear shortly, we have given the function the name `fact`.)

```
def fact (k : Nat) : Nat :=
  match k with
  | 0 => 1
  | n + 1 => (n + 1) * fact n
```

Lean can use this definition to compute `fact k` for any natural number `k`. The `match` statement tells Lean to try to match the input `k` with one of the two patterns `0` and `n + 1`, and then to use the corresponding formula after `=>` to compute `fact k`. For example, if we ask Lean for `fact 4`, it first checks if `4` matches `0`. Since it doesn't, it goes on to the next line and determines that `4` matches the pattern `n + 1`, with `n = 3`, so it uses the formula `fact 4 = 4 * fact 3`. Of course, now it must compute `fact 3`, which it does in the same way: `3` matches `n + 1` with `n = 2`, so `fact 3 = 3 * fact 2`. Continuing in this way, Lean determines that

$$\begin{aligned} \text{fact } 4 &= 4 * \text{fact } 3 = 4 * (3 * \text{fact } 2) = 4 * (3 * (2 * \text{fact } 1)) \\ &= 4 * (3 * (2 * (1 * \text{fact } 0))) = 4 * (3 * (2 * (1 * 1))) = 24. \end{aligned}$$

You can confirm this with the `#eval` command:

```
#eval fact 4 --Answer: 24
```

Of course, by now you have probably guessed why we used the name `fact` for his function: `fact k` is `k` factorial—the product of all the numbers from 1 to `k`.

This style of definition is called a *recursive* definition. If a function is defined by a recursive definition, then theorems about that function are often most easily proven by induction. For example, here is a theorem about the factorial function. It is Example 6.3.1 in *HTPI*, and we begin the Lean proof by imitating the proof in *HTPI*.

```
theorem Example_6_3_1 : ∀ n ≥ 4, fact n > 2 ^ n := by
  by_induc
  . -- Base Case
    decide
    done
  . -- Induction Step
    fix n : Nat
    assume h1 : n ≥ 4
    assume ih : fact n > 2 ^ n
    show fact (n + 1) > 2 ^ (n + 1) from
      calc fact (n + 1)
        _ = (n + 1) * fact n := by rfl
        _ > (n + 1) * 2 ^ n := sorry
        _ > 2 * 2 ^ n := sorry
        _ = 2 ^ (n + 1) := by ring
```

```
done
done
```

There are two steps in the calculational proof at the end that require justification. The first says that $(n + 1) * \text{fact } n > (n + 1) * 2^n$, which should follow from the inductive hypothesis $\text{ih} : \text{fact } n > 2^n$ by multiplying both sides by $n + 1$. Is there a theorem that would justify this inference?

This may remind you of a step in [Example_6_1_3](#) where we used the theorem `Nat.mul_le_mul_right`, which says $\forall \{n m : \mathbb{N}\} (k : \mathbb{N}), n \leq m \rightarrow n * k \leq m * k$. Our situation in this example is similar, but it involves a strict inequality ($>$ rather than \geq) and it involves multiplying on the left rather than the right. Many theorems about inequalities in Lean's library contain either `le` (for "less than or equal to") or `lt` (for "less than") in their names, but they can also be used to prove statements involving `\geq` or `$>$` . Perhaps the theorem we need is named something like `Nat.mul_lt_mul_left`. If you type `#check @Nat.mul_lt_mul_` into VS Code, a pop-up window will appear listing several theorems that begin with `Nat.mul_lt_mul_`. There is no `Nat.mul_lt_mul_left`, but there is a theorem called `Nat.mul_lt_mul_of_pos_left`, and its meaning is

```
@Nat.mul_lt_mul_of_pos_left : ∀ {n m k : \mathbb{N}},  
n < m → k > 0 → k * n < k * m
```

Lean has correctly reminded us that, to multiply both sides of a strict inequality by a number k , we need to know that $k > 0$. So in our case, we'll need to prove that $n + 1 > 0$. Once we have that, we can use the theorem `Nat.mul_lt_mul_of_pos_left` to eliminate the first `sorry`.

The second `sorry` is similar: $(n + 1) * 2^n > 2 * 2^n$ should follow from $n + 1 > 2$ and $2^n > 0$, and you can verify that the theorem that will justify this inference is `Nat.mul_lt_mul_of_pos_right`.

So we have three inequalities that we need to prove before we can justify the steps of the calculational proof: $n + 1 > 0$, $n + 1 > 2$, and $2^n > 0$. We'll insert `have` steps before the calculational proof to assert these three inequalities. If you try it, you'll find that `linarith` can prove the first two, but not the third.

How can we prove $2^n > 0$? It is often helpful to think about whether there is a general principle that is behind a statement we are trying to prove. In our case, the inequality $2^n > 0$ is an instance of the general fact that if m and n are any natural numbers with $m > 0$, then $m^n > 0$. Maybe that fact is in Lean's library:

```
example (m n : Nat) (h : m > 0) : m^n > 0 := by apply?
```

The `apply?` tactic comes up with exact `Nat.pos_pow_of_pos n h`, and `#check @Nat.pos_pow_of_pos` gives the result

```
@Nat.pos_pow_of_pos : ∀ {n : ℕ} (m : ℕ), 0 < n → 0 < n ^ m
```

That means that we can use `Nat.pos_pow_of_pos` to prove $2^n > 0$, but first we'll need to prove that $2 > 0$. We now have all the pieces we need; putting them together leads to this proof:

```
theorem Example_6_3_1 : ∀ n ≥ 4, fact n > 2 ^ n := by
  by_induc
  . -- Base Case
    decide
    done
  . -- Induction Step
    fix n : Nat
    assume h1 : n ≥ 4
    assume ih : fact n > 2 ^ n
    have h2 : n + 1 > 0 := by linarith
    have h3 : n + 1 > 2 := by linarith
    have h4 : 2 > 0 := by linarith
    have h5 : 2 ^ n > 0 := Nat.pos_pow_of_pos n h4
    show fact (n + 1) > 2 ^ (n + 1) from
      calc fact (n + 1)
        _ = (n + 1) * fact n := by rfl
        _ > (n + 1) * 2 ^ n := Nat.mul_lt_mul_of_pos_left ih h2
        _ > 2 * 2 ^ n := Nat.mul_lt_mul_of_pos_right h3 h5
        _ = 2 ^ (n + 1) := by ring
    done
  done
```

But there is an easier way. Look at the two “ $>$ ” steps in the calculational proof at the end of `Example_6_3_1`. In both cases, we took a known relationship between two quantities and did something to both sides that preserved the relationship. In the first case, the known relationship was `ih : fact n > 2 ^ n`, and we multiplied both sides by `n + 1` on the left; in the second, the known relationship was `h3 : n + 1 > 2`, and we multiplied both sides by `2 ^ n` on the right. To justify these steps, we had to find the right theorems in Lean's library, and we ended up needing auxiliary positivity facts: `h2 : n + 1 > 0` in the first case and `h5 : 2 ^ n > 0` in the second. There is a tactic that can simplify these steps: if `h` is a proof of a statement asserting a relationship between two quantities, then the tactic `rel [h]` will attempt to prove any statement obtained from that relationship by applying the same operation to both sides. The tactic will try to find a theorem in Lean's library that says that the operation preserves the relationship, and if the theorem requires auxiliary positivity facts, it will try to prove those facts as well. The `rel` tactic doesn't always succeed, but when it does, it saves you the trouble of searching through the library for the necessary theorems. In this case, the tactic allows us to give a much simpler proof of `Example_6_3_1`:

```
theorem Example_6_3_1 : ∀ n ≥ 4, fact n > 2 ^ n := by
  by_induc
  · -- Base Case
    decide
    done
  · -- Induction Step
    fix n : Nat
    assume h1 : n ≥ 4
    assume ih : fact n > 2 ^ n
    have h2 : n + 1 > 2 := by linarith
    show fact (n + 1) > 2 ^ (n + 1) from
      calc fact (n + 1)
        _ = (n + 1) * fact n := by rfl
        _ > (n + 1) * 2 ^ n := by rel [ih]
        _ > 2 * 2 ^ n := by rel [h2]
        _ = 2 ^ (n + 1) := by ring
    done
done
```

The next example in *HTPI* is a proof of one of the laws of exponents: $a^m \cdot a^n = a^{m+n}$. Lean's definition of exponentiation with natural number exponents is recursive. The definitions Lean uses are essentially as follows:

```
--For natural numbers b and k, b ^ k = nat_pow b k:
def nat_pow (b k : Nat) : Nat :=
  match k with
  | 0 => 1
  | n + 1 => (nat_pow b n) * b

--For a real number b and a natural number k, b ^ k = real_pow b k:
def real_pow (b : Real) (k : Nat) : Real :=
  match k with
  | 0 => 1
  | n + 1 => (real_pow b n) * b
```

Let's prove the addition law for exponents:

```
theorem Example_6_3_2_cheating : ∀ (a : Real) (m n : Nat),
  a ^ (m + n) = a ^ m * a ^ n := by
  fix a : Real; fix m : Nat; fix n : Nat
  ring
done
```

Well, that wasn't really fair. The `ring` tactic knows the laws of exponents, so it has no trouble proving this theorem. But we want to know *why* the law holds, so let's see if we can prove it without using `ring`. The following proof is essentially the same as the proof in *HTPI*:

```
theorem Example_6_3_2 : ∀ (a : Real) (m n : Nat),
  a ^ (m + n) = a ^ m * a ^ n := by
  fix a : Real; fix m : Nat
  --Goal : ∀ (n : Nat), a ^ (m + n) = a ^ m * a ^ n
  by_induc
  · -- Base Case
    show a ^ (m + 0) = a ^ m * a ^ 0 from
    calc a ^ (m + 0)
      _ = a ^ m := by rfl
      _ = a ^ m * 1 := (mul_one (a ^ m)).symm
      _ = a ^ m * a ^ 0 := by rfl
    done
  · -- Induction Step
    fix n : Nat
    assume ih : a ^ (m + n) = a ^ m * a ^ n
    show a ^ (m + (n + 1)) = a ^ m * a ^ (n + 1) from
    calc a ^ (m + (n + 1))
      _ = a ^ ((m + n) + 1) := by rw [add_assoc]
      _ = a ^ (m + n) * a := by rfl
      _ = (a ^ m * a ^ n) * a := by rw [ih]
      _ = a ^ m * (a ^ n * a) := by rw [mul_assoc]
      _ = a ^ m * (a ^ (n + 1)) := by rfl
    done
  done
```

Finally, we'll prove the theorem in Example 6.3.4 of *HTPI*, which again involves exponentiation with natural number exponents. Here's the beginning of the proof:

Lean File

```
theorem Example_6_3_4 : ∀ (x : Real), x > -1 →
  ∀ (n : Nat), (1 + x) ^ n ≥ 1 + n * x := by
  fix x : Real
  assume h1 : x > -1
  done
```

Tactic State in Infoview

```
x : ℝ
h1 : x > -1
↑ ∀ (n : ℕ),
(1 + x) ^ n ≥
1 + ↑n * x
```

Look carefully at the goal in the tactic state. Why is there a `↑` before the last `n`? The reason has to do with types. The variable `x` has type `Real` and `n` has type `Nat`, so how can Lean multiply `n` by `x`? Remember, in Lean, the natural numbers are not a subset of the real numbers. The

two types are completely separate, but for each natural number, there is a corresponding real number. To multiply n by x , Lean had to convert n to the corresponding real number, through a process called *coercion*. The notation $\uparrow n$ denotes the result of *coercing* (or *casting*) n to another type—in this case, `Real`. Since $\uparrow n$ and x are both real numbers, Lean can use the multiplication operation on the real numbers to multiply them. (To type \uparrow in VSCode, type `\uparrowarrow`, or just `\u.`)

As we will see, the need for coercion in this example will make the proof a bit more complicated, because we'll need to use some theorems about coercions. Theorems about coercion of natural numbers to some other type often have names that start `Nat.cast`.

Continuing with the proof, since exponentiation is defined recursively, let's try mathematical induction:

Lean File

```
theorem Example_6_3_4 : ∀ (x : Real), x > -1 →
  ∀ (n : Nat), (1 + x) ^ n ≥ 1 + n * x := by
  fix x : Real
  assume h1 : x > -1
  by_induc
  · -- Base Case
    done
  · -- Induction Step
    done
done
```

Tactic State in Infoview

```
▼ case Base_Case
x : ℝ
h1 : x > -1
↑(1 + x) ^ 0 ≥
1 + ↑0 * x
```

You might think that `linarith` could prove the goal for the base case, but it can't. The problem is the $\uparrow 0$, which denotes the result of coercing the natural number 0 to a real number. Of course, that should be the real number 0 , but is it? Yes, but the `linarith` tactic doesn't know that. The theorem `Nat.cast_zero` says that $\uparrow 0 = 0$ (where the 0 on the right side of the equation is the *real number* 0), so the tactic `rewrite [Nat.cast_zero]` will convert $\uparrow 0$ to 0 . After that step, `linarith` can complete the proof of the base case, and we can start on the induction step.

Lean File

```
theorem Example_6_3_4 : ∀ (x : Real), x > -1 →
  ∀ (n : Nat), (1 + x) ^ n ≥ 1 + n * x := by
  fix x : Real
  assume h1 : x > -1
  by_induc
  · -- Base Case
    rewrite [Nat.cast_zero]
    linarith
    done
  · -- Induction Step
    fix n : Nat
    assume ih : (1 + x) ^ n ≥ 1 + n * x
    done
done
```

Tactic State in Infoview

```
▼case Induction_Step
x : ℝ
h1 : x > -1
n : ℕ
ih : (1 + x) ^ n ≥
  1 + ↑n * x
↑(1 + x) ^ (n + 1) ≥
  1 + ↑(n + 1) * x
```

Once again, there's a complication caused by coercion. The inductive hypothesis talks about $\uparrow n$, but the goal involves $\uparrow(n + 1)$. What is the relationship between these? Surely it should be the case that $\uparrow(n + 1) = \uparrow n + 1$; that is, the result of coercing the natural number $n + 1$ to a real number should be one larger than the result of coercing n to a real number. The theorem `Nat.cast_succ` says exactly that, so `rewrite [Nat.cast_succ]` will change the $\uparrow(n + 1)$ in the goal to $\uparrow n + 1$. (The number $n + 1$ is sometimes called the *successor* of n , and `succ` is short for “successor.”) With that change, we can continue with the proof. The following proof is modeled on the proof in *HTPI*.

```
theorem Example_6_3_4 : ∀ (x : Real), x > -1 →
  ∀ (n : Nat), (1 + x) ^ n ≥ 1 + n * x := by
  fix x : Real
  assume h1 : x > -1
  by_induc
  · -- Base Case
    rewrite [Nat.cast_zero]
    linarith
    done
  · -- Induction Step
    fix n : Nat
    assume ih : (1 + x) ^ n ≥ 1 + n * x
    rewrite [Nat.cast_succ]
    show (1 + x) ^ (n + 1) ≥ 1 + (n + 1) * x from
      calc (1 + x) ^ (n + 1)
        _ = (1 + x) ^ n * (1 + x) := by rfl
        _ ≥ (1 + n * x) * (1 + x) := sorry
```

```

_ = 1 + n * x + x + n * x ^ 2 := by ring
_ ≥ 1 + n * x + x + 0 := sorry
_ = 1 + (n + 1) * x := by ring
done
done

```

Note that in the calculational proof, each n or $n + 1$ that is multiplied by x is really $\uparrow n$ or $\uparrow n + 1$, but we don't need to say so explicitly; Lean fills in coercions automatically when they are required.

All that's left is to replace the two occurrences of `sorry` with justifications. The first `sorry` step should follow from the inductive hypothesis by multiplying both sides by $1 + x$, so a natural attempt to justify it would be `by rel [ih]`. Unfortunately, we get an error message saying that `rel` failed. The error message tells us that `rel` needed to know that $0 \leq 1 + x$, and it was unable to prove it, so we'll have to provide a proof of that statement ourselves. Fortunately, `linarith` can handle it (deducing it from `h1 : x > -1`), and once we fill in that additional step, the `rel` tactic succeeds.

```

theorem Example_6_3_4 : ∀ (x : Real), x > -1 →
  ∀ (n : Nat), (1 + x) ^ n ≥ 1 + n * x := by
  fix x : Real
  assume h1 : x > -1
  by_reduc
  · -- Base Case
    rewrite [Nat.cast_zero]
    linarith
    done
  · -- Induction Step
    fix n : Nat
    assume ih : (1 + x) ^ n ≥ 1 + n * x
    rewrite [Nat.cast_succ]
    have h2 : 0 ≤ 1 + x := by linarith
    show (1 + x) ^ (n + 1) ≥ 1 + (n + 1) * x from
      calc (1 + x) ^ (n + 1)
        _ = (1 + x) ^ n * (1 + x) := by rfl
        _ ≥ (1 + n * x) * (1 + x) := by rel [ih]
        _ = 1 + n * x + x + n * x ^ 2 := by ring
        _ ≥ 1 + n * x + x + 0 := sorry
        _ = 1 + (n + 1) * x := by ring
    done
done

```

6.3. Recursion

For the second `sorry` step, we'll need to know that $n * x^2 \geq 0$. To prove it, we start with the fact that the square of any real number is nonnegative:

```
@sq_nonneg : ∀ {α : Type u_1} [inst : LinearOrderedSemiring α]
[inst_1 : ExistsAddOfLE α]
(a : α), 0 ≤ a ^ 2
```

As usual, we don't need to pay much attention to the implicit arguments; what is important is the last line, which tells us that `sq_nonneg x` is a proof of $x^2 \geq 0$. To get $n * x^2 \geq 0$ we just have to multiply both sides by n , which we can justify with the `rel` tactic, and then one more application of `rel` will handle the remaining `sorry`. Here is the complete proof:

```
theorem Example_6_3_4 : ∀ (x : Real), x > -1 →
  ∀ (n : Nat), (1 + x) ^ n ≥ 1 + n * x := by
  fix x : Real
  assume h1 : x > -1
  by_induc
  · -- Base Case
    rewrite [Nat.cast_zero]
    linarith
    done
  · -- Induction Step
    fix n : Nat
    assume ih : (1 + x) ^ n ≥ 1 + n * x
    rewrite [Nat.cast_succ]
    have h2 : 0 ≤ 1 + x := by linarith
    have h3 : x ^ 2 ≥ 0 := sq_nonneg x
    have h4 : n * x ^ 2 ≥ 0 :=
      calc n * x ^ 2
        _ ≥ n * 0 := by rel [h3]
        _ = 0 := by ring
    show (1 + x) ^ (n + 1) ≥ 1 + (n + 1) * x from
      calc (1 + x) ^ (n + 1)
        _ = (1 + x) ^ n * (1 + x) := by rfl
        _ ≥ (1 + n * x) * (1 + x) := by rel [ih]
        _ = 1 + n * x + x + n * x ^ 2 := by ring
        _ ≥ 1 + n * x + x + 0 := by rel [h4]
        _ = 1 + (n + 1) * x := by ring
    done
  done
```

Before ending this section, we'll return to a topic left unexplained before. We can now describe how `Sum i from k to n, f i` is defined. The key is a function `sum_seq`, which is defined by recursion:

```
def sum_seq {A : Type} [AddZeroClass A]
  (m k : Nat) (f : Nat → A) : A :=
  match m with
  | 0 => 0
  | n + 1 => sum_seq n k f + f (k + n)
```

To get an idea of what this definition means, let's try evaluating `sum_seq 3 k f`:

$$\begin{aligned} \text{sum_seq } 3 \text{ } k \text{ } f &= \text{sum_seq } 2 \text{ } k \text{ } f + f (k + 2) \\ &= \text{sum_seq } 1 \text{ } k \text{ } f + f (k + 1) + f (k + 2) \\ &= \text{sum_seq } 0 \text{ } k \text{ } f + f (k + 0) + f (k + 1) + f (k + 2) \\ &= 0 + f (k + 0) + f (k + 1) + f (k + 2) \\ &= f k + f (k + 1) + f (k + 2). \end{aligned}$$

So `sum_seq 3 k f` adds up three consecutive values of `f`, starting with `f k`. More generally, `sum_seq n k f` adds up a sequence of `n` consecutive values of `f`, starting with `f k`. (The implicit arguments say that the type of the values of `f` can be any type for which `+` and `0` make sense.) The notation `Sum i from k to n, f i` is now defined to be a shorthand for `sum_seq (n + 1 - k) k f`. We'll leave it to you to puzzle out why that gives the desired result.

Exercises

1. `theorem Exercise_6_3_4 : ∀ (n : Nat), 3 * (Sum i from 0 to n, (2 * i + 1) ^ 2) = (n + 1) * (2 * n + 1) * (2 * n + 3) := sorry`
2. `theorem Exercise_6_3_7b (f : Nat → Real) (c : Real) : ∀ (n : Nat), Sum i from 0 to n, c * f i = c * Sum i from 0 to n, f i := sorry`
3. `theorem fact_pos : ∀ (n : Nat), fact n ≥ 1 := sorry`
4. `--Hint: Use the theorem fact_pos from the previous exercise.`
`theorem Exercise_6_3_13a (k : Nat) : ∀ (n : Nat), fact (k ^ 2 + n) ≥ k ^ (2 * n) := sorry`
5. `--Hint: Use the theorem in the previous exercise.`
`--You may find it useful to first prove a lemma:`
`--∀ (k : Nat), 2 * k ^ 2 + 1 ≥ k`
`theorem Exercise_6_3_13b (k : Nat) : ∀ n ≥ 2 * k ^ 2, fact n ≥ k ^ n := sorry`
6. A sequence is defined recursively as follows:

```
def seq_6_3_15 (k : Nat) : Int :=
  match k with
  | 0 => 0
  | n + 1 => 2 * seq_6_3_15 n + n
```

Prove the following theorem about this sequence:

```
theorem Exercise_6_3_15 : ∀ (n : Nat),
  seq_6_3_15 n = 2 ^ n - n - 1 := sorry
```

7. A sequence is defined recursively as follows:

```
def seq_6_3_16 (k : Nat) : Nat :=
  match k with
  | 0 => 2
  | n + 1 => (seq_6_3_16 n) ^ 2
```

Find a formula for $\text{seq}_6_3_16 n$. Fill in the blank in the theorem below with your formula and then prove the theorem.

```
theorem Exercise_6_3_16 : ∀ (n : Nat),
  seq_6_3_16 n = ___ := sorry
```

6.4. Strong Induction

In the induction step of a proof by mathematical induction, we prove that a natural number has some property from the assumption that the previous number has the property. Section 6.4 of *HTPI* introduces a version of mathematical induction in which we get to assume that *all* smaller numbers have the property. Since this is a stronger assumption, this version of induction is called *strong induction*. Here is how strong induction works (*HTPI* p. 304):

To prove a goal of the form $\forall (n : \text{Nat}), P n$:

Prove that $\forall (n : \text{Nat}), (\forall n_{-1} < n, P n_{-1}) \rightarrow P n$.

To write a proof by strong induction in Lean, we use the tactic `by_strong_induc`, whose effect on the tactic state can be illustrated as follows.

6.4. Strong Induction

Tactic State Before Using Strategy

```
:  
↑  $\forall (n : \text{Nat}), P n$ 
```

Tactic State After Using Strategy

```
:  
↑  $\forall (n : \text{Nat}),$   
 $(\forall n_1 < n, P n_1) \rightarrow P n$ 
```

To illustrate this, we begin with Example 6.4.1 of *HTPI*.

```
theorem Example_6_4_1 :  $\forall m > 0, \forall (n : \text{Nat}),$   
 $\exists (q r : \text{Nat}), n = m * q + r \wedge r < m$ 
```

Imitating the strategy of the proof in *HTPI*, we let m be an arbitrary natural number, assume $m > 0$, and then prove the statement $\forall (n : \text{Nat}), \exists (q r : \text{Nat}), n = m * q + r \wedge r < m$ by strong induction. That means that after introducing an arbitrary natural number n , we assume the inductive hypothesis, which says $\forall n_1 < n, \exists (q r : \text{Nat}), n_1 = m * q + r \wedge r < m$.

```
theorem Example_6_4_1 :  $\forall m > 0, \forall (n : \text{Nat}),$   
 $\exists (q r : \text{Nat}), n = m * q + r \wedge r < m := \text{by}$   
 $\text{fix } m : \text{Nat}$   
 $\text{assume } h1 : m > 0$   
 $\text{by\_strong\_induc}$   
 $\text{fix } n : \text{Nat}$   
 $\text{assume } ih : \forall n_1 < n, \exists (q r : \text{Nat}), n_1 = m * q + r \wedge r < m$   
 $\text{done}$ 
```

Our goal now is to prove that $\exists (q r : \text{Nat}), n = m * q + r \wedge r < m$. Although strong induction does not require a base case, it is not uncommon for proofs by strong induction to involve reasoning by cases. The proof in *HTPI* uses cases based on whether or not $n < m$. If $n < m$, then the proof is easy: the numbers $q = 0$ and $r = n$ clearly have the required properties. If $n > m$, then we can write n as $n = k + m$, for some natural number k . Since $m > 0$, we have $k < n$, so we can apply the inductive hypothesis to k . Notice that if $m > 1$, then k is not the number immediately preceding n ; that's why this proof uses strong induction rather than ordinary induction.

How do we come up with the number k in the previous paragraph? We'll use a theorem from Lean's library. There are two slightly different versions of the theorem—notice that the first ends with $m + k$ and the second ends with $k + m$:

```
@Nat.exists_eq_add_of_le :  $\forall \{m n : \mathbb{N}\}, m \leq n \rightarrow \exists (k : \mathbb{N}), n = m + k$   
@Nat.exists_eq_add_of_le' :  $\forall \{m n : \mathbb{N}\}, m \leq n \rightarrow \exists (k : \mathbb{N}), n = k + m$ 
```

6.4. Strong Induction

We'll use the second version in our proof.

```
theorem Example_6_4_1 :  $\forall m > 0, \forall (n : \text{Nat})$ ,  

     $\exists (q r : \text{Nat}), n = m * q + r \wedge r < m :=$  by  

fix m : Nat  

assume h1 : m > 0  

by_strong_induc  

fix n : Nat  

assume ih :  $\forall n_1 < n, \exists (q r : \text{Nat}), n_1 = m * q + r \wedge r < m$   

by_cases h2 : n < m  

• -- Case 1. h2 : n < m  

  apply Exists.intro 0  

  apply Exists.intro n      --Goal : n = m * 0 + n  $\wedge$  n < m  

  apply And.intro _ h2  

  ring  

  done  

• -- Case 2. h2 :  $\neg n < m$   

  have h3 : m ≤ n := by linarith  

  obtain (k : Nat) (h4 : n = k + m) from Nat.exists_eq_add_of_le' h3  

  have h5 : k < n := by linarith  

  have h6 :  $\exists (q r : \text{Nat}), k = m * q + r \wedge r < m :=$  ih k h5  

  obtain (q' : Nat)  

    (h7 :  $\exists (r : \text{Nat}), k = m * q' + r \wedge r < m$ ) from h6  

  obtain (r' : Nat) (h8 : k = m * q' + r'  $\wedge$  r' < m) from h7  

  apply Exists.intro (q' + 1)  

  apply Exists.intro r'      --Goal : n = m * (q' + 1) + r'  $\wedge$  r' < m  

  apply And.intro _ h8.right  

  show n = m * (q' + 1) + r' from  

    calc n  

    _ = k + m := h4  

    _ = m * q' + r' + m := by rw [h8.left]  

    _ = m * (q' + 1) + r' := by ring  

  done  

done
```

The numbers q and r in `Example_6_4_1` are called the *quotient* and *remainder* when n is divided by m . Lean knows how to compute these numbers: if n and m are natural numbers, then in Lean, n / m denotes the quotient when n is divided by m , and $n \% m$ denotes the remainder. (The number $n \% m$ is also sometimes called n *modulo* m , or n *mod* m .) And Lean knows theorems stating that these numbers have the properties specified in `Example_6_4_1`:

`@Nat.div_add_mod : $\forall (m n : \mathbb{N}), n * (m / n) + m \% n = m$`

```
@Nat.mod_lt : ∀ (x : ℕ) {y : ℕ}, y > 0 → x % y < y
```

By the way, although we are unlikely to want to use the notation $n / 0$ or $n \% 0$, Lean uses the definitions $n / 0 = 0$ and $n \% 0 = n$. As a result, the equation $n * (m / n) + m \% n = m$ is true even if $n = 0$. That's why the theorem `Nat.div_add_mod` doesn't include a requirement that $n > 0$. It is important to keep in mind that division of natural numbers is not the same as division of real numbers. For example, dividing the natural number 5 by the natural number 2 gives a quotient of 2 (with a remainder of 1), so $(5 : Nat) / (2 : Nat)$ is 2, but $(5 : Real) / (2 : Real)$ is 2.5.

There is also a strong form of recursion. As an example of this, here is a recursive definition of a sequence of numbers called the *Fibonacci numbers*:

```
def Fib (n : Nat) : Nat :=
  match n with
  | 0 => 0
  | 1 => 1
  | k + 2 => Fib k + Fib (k + 1)
```

Notice that the formula for `Fib (k + 2)` involves the *two* previous values of `Fib`, not just the immediately preceding value. That is the sense in which the recursion is *strong*. Not surprisingly, theorems about the Fibonacci numbers are often proven by induction—either ordinary or strong. We'll illustrate this with a proof by strong induction that $\forall (n : \text{Nat})$, $\text{Fib } n < 2^n$. This time we'll need to treat the cases $n = 0$ and $n = 1$ separately, since these values are treated separately in the definition of `Fib n`. And we'll need to know that if n doesn't fall into either of those cases, then it falls into the third case: $n = k + 2$ for some natural number k . Since similar ideas will come up several times in the rest of this book, it will be useful to begin by proving lemmas that will help with this kind of reasoning.

We'll need two theorems from Lean's library, the second of which has two slightly different versions:

```
@Nat.pos_of_ne_zero : ∀ {n : ℕ}, n ≠ 0 → 0 < n

@lt_of_le_of_ne : ∀ {α : Type u_1} [inst : PartialOrder α] {a b : α},
  a ≤ b → a ≠ b → a < b

@lt_of_le_of_ne' : ∀ {α : Type u_1} [inst : PartialOrder α] {a b : α},
  a ≤ b → b ≠ a → a < b
```

If we have $h1 : n \neq 0$, then `Nat.pos_of_ne_zero h1` is a proof of $0 < n$. But for natural numbers a and b , Lean treats $a < b$ as meaning the same thing as $a + 1 \leq b$, so this is also a proof of $1 \leq n$. If we also have $h2 : n \neq 1$, then we can use `lt_of_le_of_ne'` to conclude $1 < n$, which is definitionally equal to $2 \leq n$. Combining this reasoning with the theorem

6.4. Strong Induction

`Nat.exists_eq_add_of_le'`, which we used in the last example, we can prove two lemmas that will be helpful for reasoning in which the first one or two natural numbers have to be treated separately.

```
lemma exists_eq_add_one_of_ne_zero {n : Nat}
  (h1 : n ≠ 0) : ∃ (k : Nat), n = k + 1 := by
  have h2 : 1 ≤ n := Nat.pos_of_ne_zero h1
  show ∃ (k : Nat), n = k + 1 from Nat.exists_eq_add_of_le' h2
  done

theorem exists_eq_add_two_of_ne_zero_one {n : Nat}
  (h1 : n ≠ 0) (h2 : n ≠ 1) : ∃ (k : Nat), n = k + 2 := by
  have h3 : 1 ≤ n := Nat.pos_of_ne_zero h1
  have h4 : 2 ≤ n := lt_of_le_of_ne' h3 h2
  show ∃ (k : Nat), n = k + 2 from Nat.exists_eq_add_of_le' h4
  done
```

With this preparation, we can present the proof:

```
example : ∀ (n : Nat), Fib n < 2 ^ n := by
  by_strong_induc
  fix n : Nat
  assume ih : ∀ n_1 < n, Fib n_1 < 2 ^ n_1
  by_cases h1 : n = 0
  · -- Case 1. h1 : n = 0
    rewrite [h1] --Goal : Fib 0 < 2 ^ 0
    decide
    done
  · -- Case 2. h1 : ¬n = 0
    by_cases h2 : n = 1
    · -- Case 2.1. h2 : n = 1
      rewrite [h2]
      decide
      done
    · -- Case 2.2. h2 : ¬n = 1
      obtain (k : Nat) (h3 : n = k + 2) from
        exists_eq_add_two_of_ne_zero_one h1 h2
      have h4 : k < n := by linarith
      have h5 : Fib k < 2 ^ k := ih k h4
      have h6 : k + 1 < n := by linarith
      have h7 : Fib (k + 1) < 2 ^ (k + 1) := ih (k + 1) h6
      rewrite [h3] --Goal : Fib (k + 2) < 2 ^ (k + 2)
```

```

show Fib (k + 2) < 2 ^ (k + 2) from
calc Fib (k + 2)
  _ = Fib k + Fib (k + 1) := by rfl
  _ < 2 ^ k + Fib (k + 1) := by rel [h5]
  _ < 2 ^ k + 2 ^ (k + 1) := by rel [h7]
  _ ≤ 2 ^ k + 2 ^ (k + 1) + 2 ^ k := by linarith
  _ = 2 ^ (k + 2) := by ring
done
done
done

```

As with ordinary induction, strong induction can be useful for proving statements that do not at first seem to have the form $\forall (n : \text{Nat})$, ... To illustrate this, we'll prove the *well-ordering principle*, which says that if a set $S : \text{Set Nat}$ is nonempty, then it has a smallest element. We'll prove the contrapositive: if S has no smallest element, then it is empty. To say that S is empty means $\forall (n : \text{Nat})$, $n \notin S$, and that's the statement to which we will apply strong induction.

```

theorem well_ord_princ (S : Set Nat) : ( $\exists (n : \text{Nat})$ ,  $n \in S$ )  $\rightarrow$ 
   $\exists n \in S$ ,  $\forall m \in S$ ,  $n \leq m$  := by
contrapos
assume h1 :  $\neg\exists n \in S$ ,  $\forall m \in S$ ,  $n \leq m$ 
quant_neg
by_strong_induc
fix n : Nat
assume ih :  $\forall n_1 < n$ ,  $n_1 \notin S$  --Goal :  $n \notin S$ 
contradict h1 with h2      --h2 : n ∈ S
--Goal :  $\exists n \in S$ ,  $\forall m \in S$ ,  $n \leq m$ 
apply Exists.intro n      --Goal : n ∈ S  $\wedge$   $\forall m \in S$ ,  $n \leq m$ 
apply And.intro h2        --Goal :  $\forall m \in S$ ,  $n \leq m$ 
fix m : Nat
assume h3 : m ∈ S
have h4 : m < n  $\rightarrow$  m ∉ S := ih m
contrapos at h4           --h4 : m ∈ S  $\rightarrow$   $\neg m < n$ 
have h5 :  $\neg m < n$  := h4 h3
linarith
done

```

Section 6.4 of *HTPI* ends with an example of an application of the well-ordering principle. The example gives a proof that $\sqrt{2}$ is irrational. If $\sqrt{2}$ were rational, then there would be natural numbers p and q such that $q \neq 0$ and $p/q = \sqrt{2}$, and therefore $p^2 = 2q^2$. So we can

6.4. Strong Induction

prove that $\sqrt{2}$ is irrational by showing that there do not exist natural numbers p and q such that $q \neq 0$ and $p^2 = 2q^2$.

The proof uses a definition from the exercises of Section 6.1:

```
def nat_even (n : Nat) : Prop := ∃ (k : Nat), n = 2 * k
```

We will also use the following lemma, whose proof we leave as an exercise for you:

```
lemma sq_even_iff_even (n : Nat) : nat_even (n * n) ↔ nat_even n := sorry
```

And we'll need another theorem that we haven't seen before:

```
@mul_left_cancel_iff_of_pos : ∀ {α : Type u_1} {a b c : α}
  [inst : MulZeroClass α] [inst_1 : PartialOrder α]
  [inst_2 : PosMulReflectLE α],
  0 < a → (a * b = a * c ↔ b = c)
```

To show that $\sqrt{2}$ is irrational, we will prove the statement

```
¬∃ (q p : Nat), p * p = 2 * (q * q) ∧ q ≠ 0
```

We proceed by contradiction. If this statement were false, then the set

```
S = {q : Nat | ∃ (p : Nat), p * p = 2 * (q * q) ∧ q ≠ 0}
```

would be nonempty, and therefore, by the well-ordering principle, it would have a smallest element. We then show that this leads to a contradiction. Here is the proof.

```
theorem Theorem_6_4_5 :
  ¬∃ (q p : Nat), p * p = 2 * (q * q) ∧ q ≠ 0 := by
  set S : Set Nat :=
  {q : Nat | ∃ (p : Nat), p * p = 2 * (q * q) ∧ q ≠ 0}
  by_contra h1
  have h2 : ∃ (q : Nat), q ∈ S := h1
  have h3 : ∃ q ∈ S, ∀ r ∈ S, q ≤ r := well_ord_princ S h2
  obtain (q : Nat) (h4 : q ∈ S ∧ ∀ r ∈ S, q ≤ r) from h3
  have qinS : q ∈ S := h4.left
  have qleast : ∀ r ∈ S, q ≤ r := h4.right
  define at qinS --qinS : ∃ (p : Nat), p * p = 2 * (q * q) ∧ q ≠ 0
  obtain (p : Nat) (h5 : p * p = 2 * (q * q) ∧ q ≠ 0) from qinS
  have pqsqrt2 : p * p = 2 * (q * q) := h5.left
  have qne0 : q ≠ 0 := h5.right
  have h6 : nat_even (p * p) := Exists.intro (q * q) pqsqrt2
```

```

rewrite [sq_even_iff_even p] at h6      --h6 : nat_even p
obtain (p' : Nat) (p'halfp : p = 2 * p') from h6
have h7 : 2 * (2 * (p' * p')) = 2 * (q * q) := by
  rewrite [←pqsqrt2, p'halfp]
  ring
  done
have h8 : 2 > 0 := by decide
rewrite [mul_left_cancel_iff_of_pos h8] at h7
--h7 : 2 * (p' * p') = q * q
have h9 : nat_even (q * q) := Exists.intro (p' * p') h7.symm
rewrite [sq_even_iff_even q] at h9      --h9 : nat_even q
obtain (q' : Nat) (q'halfq : q = 2 * q') from h9
have h10 : 2 * (p' * p') = 2 * (2 * (q' * q')) := by
  rewrite [h7, q'halfq]
  ring
  done
rewrite [mul_left_cancel_iff_of_pos h8] at h10
--h10 : p' * p' = 2 * (q' * q')
have q'ne0 : q' ≠ 0 := by
  contradict qne0 with h11
  rewrite [q'halfq, h11]
  rfl
  done
have q'inS : q' ∈ S := Exists.intro p' (And.intro h10 q'ne0)
have qleq' : q ≤ q' := qleast q' q'inS
rewrite [q'halfq] at qleq'      --qleq' : 2 * q' ≤ q'
contradict q'ne0
linarith
done

```

Exercises

- Hint: Use Exercise_6_1_16a1 and Exercise_6_1_16a2.
--from the exercises of Section 6.1.

```

lemma sq_even_iff_even (n : Nat) :
  nat_even (n * n) ↔ nat_even n := sorry

```
- This theorem proves that the square root of 6 is irrational

```

theorem Exercise_6_4_4a :
  ¬∃ (q p : Nat), p * p = 6 * (q * q) ∧ q ≠ 0 := sorry

```

3. `theorem Exercise_6_4_5 :
 ∀ n ≥ 12, ∃ (a b : Nat), 3 * a + 7 * b = n := sorry`
4. `theorem Exercise_6_4_7a : ∀ (n : Nat),
 (Sum i from 0 to n, Fib i) + 1 = Fib (n + 2) := sorry`
5. `theorem Exercise_6_4_7c : ∀ (n : Nat),
 Sum i from 0 to n, Fib (2 * i + 1) = Fib (2 * n + 2) := sorry`
6. `theorem Exercise_6_4_8a : ∀ (m n : Nat),
 Fib (m + n + 1) = Fib m * Fib n + Fib (m + 1) * Fib (n + 1) := sorry`
7. `theorem Exercise_6_4_8d : ∀ (m k : Nat), Fib m | Fib (m * k) := sorry`

Hint for #7: Let m be an arbitrary natural number, and then use induction on k . For the induction step, you must prove $\text{Fib } m \mid \text{Fib } (m * (k + 1))$. If $m = 0 \vee k = 0$, then this is easy. If not, then use `exists_eq_add_one_of_ne_zero` to obtain a natural number j such that $m * k = j + 1$, and therefore $m * (k + 1) = j + m + 1$, and then apply `Exercise_6_4_8a`.

8. `def Fib_like (n : Nat) : Nat :=
 match n with
 | 0 => 1
 | 1 => 2
 | k + 2 => 2 * (Fib_like k) + Fib_like (k + 1)

theorem Fib_like_formula : ∀ (n : Nat), Fib_like n = 2 ^ n := sorry`

9. `def triple_rec (n : Nat) : Nat :=
 match n with
 | 0 => 0
 | 1 => 2
 | 2 => 4
 | k + 3 => 4 * triple_rec k +
 6 * triple_rec (k + 1) + triple_rec (k + 2)

theorem triple_rec_formula :
 ∀ (n : Nat), triple_rec n = 2 ^ n * Fib n := sorry`

10. In this exercise you will prove that the numbers q and r in `Example_6_4_1` are unique. It is helpful to prove a lemma first.

```
lemma quot_rem_unique_lemma {m q r q' r' : Nat}  
  (h1 : m * q + r = m * q' + r') (h2 : r' < m) : q ≤ q' := sorry  
  
theorem quot_rem_unique (m q r q' r' : Nat)
```

```
(h1 : m * q + r = m * q' + r') (h2 : r < m) (h3 : r' < m) :
q = q' ∧ r = r' := sorry
```

11. Use the theorem in the previous exercise to prove the following characterization of n / m and $n \% m$.

```
theorem div_mod_char (m n q r : Nat)
  (h1 : n = m * q + r) (h2 : r < m) : q = n / m ∧ r = n \% m := sorry
```

6.5. Closures Again

Section 6.5 of *HTPI* gives one more application of recursion and induction: another proof of the existence of closures of sets under functions. Recall from Section 5.4 that if $f : A \rightarrow A$ and $B : \text{Set } A$, then the *closure* of B under f is the smallest set containing B that is closed under f . In Section 5.4, we constructed the closure of B under f by taking the intersection of all sets containing B that are closed under f . In this section, we construct the closure by starting with the set B and repeatedly taking the image under f . For the motivation for this strategy, see *HTPI*; here we focus on how to carry out this strategy in Lean.

To talk about repeatedly taking the image of a set under a function, we will need a recursive definition:

```
def rep_image {A : Type} (f : A → A) (n : Nat) (B : Set A) : Set A :=
  match n with
  | 0 => B
  | k + 1 => image f (rep_image f k B)
```

According to this definition, $\text{rep_image } f \ 0 \ B = B$, $\text{rep_image } f \ 1 \ B = \text{image } f \ B$, $\text{rep_image } f \ 2 \ B = \text{image } f \ (\text{image } f \ B)$, and so on. In other words, $\text{rep_image } f \ n \ B$ is the result of starting with B and then taking the image under f n times. To make it easier to work with this definition, we state two simple theorems, both of which follow immediately from the definition.

```
theorem rep_image_base {A : Type} (f : A → A) (B : Set A) :
  rep_image f 0 B = B := by rfl

theorem rep_image_step {A : Type} (f : A → A) (n : Nat) (B : Set A) :
  rep_image f (n + 1) B = image f (rep_image f n B) := by rfl
```

We will prove that the closure of B under f is the union of the sets $\text{rep_image } f \ n \ B$. We will call this the *cumulative image* of B under f , and we define it as follows:

6.5. Closures Again

```
def cumul_image {A : Type} (f : A → A) (B : Set A) : Set A :=
{ $x : A \mid \exists (n : \mathbb{N}), x \in \text{rep\_image } f \ n \ B\}$ 
```

To prove that `cumul_image f B` is the closure of `B` under `f`, we first prove a lemma saying that if $B \subseteq D$ and D is closed under `f`, then for every natural number n , `rep_image f n B ⊆ D`. We prove it by induction.

```
lemma rep_image_sub_closed {A : Type} {f : A → A} {B D : Set A}
  ( $h1 : B \subseteq D$ ) ( $h2 : \text{closed } f \ D$ ) :
   $\forall (n : \mathbb{N}), \text{rep\_image } f \ n \ B \subseteq D := \text{by}$ 
  by_induc
  · -- Base Case
    rewrite [rep_image_base]           --Goal : B ⊆ D
    show B ⊆ D from h1
    done
  · -- Induction Step
    fix n : \mathbb{N}
    assume ih : rep_image f n B ⊆ D   --Goal : rep_image f (n + 1) B ⊆ D
    fix x : A
    assume h3 : x ∈ rep_image f (n + 1) B  --Goal : x ∈ D
    rewrite [rep_image_step] at h3
    define at h3   --h3 : ∃ x_1 ∈ rep_image f n B, f x_1 = x
    obtain (b : A) (h4 : b ∈ rep_image f n B ∧ f b = x) from h3
    rewrite [←h4.right]  --Goal : f b ∈ D
    have h5 : b ∈ D := ih h4.left
    define at h2      --h2 : ∀ x ∈ D, f x ∈ D
    show f b ∈ D from h2 b h5
    done
  done
```

With this preparation, we can now prove that `cumul_image f B` is the closure of `B` under `f`.

```
theorem Theorem_6_5_1 {A : Type} (f : A → A) (B : Set A) :
  closure f B (cumul_image f B) := by
  define
  apply And.intro
  · -- Proof that cumul_image f B ∈ {D : Set A | B ⊆ D ∧ closed f D}
    define --Goal : B ⊆ cumul_image f B ∧ closed f (cumul_image f B)
    apply And.intro
    · -- Proof that  $B \subseteq \text{cumul\_image } f \ B$ 
      fix x : A
```

```

assume h1 : x ∈ B
define    --Goal : ∃ (n : Nat), x ∈ rep_image f n B
apply Exists.intro 0
rewrite [rep_image_base] --Goal : x ∈ B
show x ∈ B from h1
done

• -- Proof that cumul_image f B closed under f
define
fix x : A
assume h1 : x ∈ cumul_image f B --Goal : f x ∈ cumul_image f B
define at h1
obtain (m : Nat) (h2 : x ∈ rep_image f m B) from h1
define    --Goal : ∃ (n : Nat), f x ∈ rep_image f n B
apply Exists.intro (m + 1) --Goal : f x ∈ rep_image f (m + 1) B
rewrite [rep_image_step] --Goal : f x ∈ image f (rep_image f m B)
define    --Goal : ∃ x_1 ∈ rep_image f m B, f x_1 = f x
apply Exists.intro x --Goal : x ∈ rep_image f m B ∧ f x = f x
apply And.intro h2
rfl
done
done

• -- Proof that cumul_image f B is smallest
fix D : Set A
assume h1 : D ∈ {D : Set A | B ⊆ D ∧ closed f D}
define at h1 --h1 : B ⊆ D ∧ closed f D
define    --Goal : ∀ {a : A}, a ∈ cumul_image f B → a ∈ D
fix x : A
assume h2 : x ∈ cumul_image f B --Goal : x ∈ D
define at h2 --h2: ∃ (n : Nat), x ∈ rep_image f n B
obtain (m : Nat) (h3 : x ∈ rep_image f m B) from h2
have h4 : rep_image f m B ⊆ D :=
  rep_image_sub_closed h1.left h1.right m
show x ∈ D from h4 h3
done
done
done

```

Exercises

- Recall the following definitions from the exercises of Section 5.4:

6.5. Closures Again

```
def closed_family {A : Type} (F : Set (A → A)) (C : Set A) : Prop :=
  ∀ f ∈ F, closed f C

def closure_family {A : Type} (F : Set (A → A)) (B C : Set A) : Prop :=
  smallestElt (sub A) C {D : Set A | B ⊆ D ∧ closed_family F D}
```

These definitions say that a set is closed under a family of functions if it is closed under all of the functions in the family, and the closure of a set B under a family of functions is the smallest set containing B that is closed under the family.

In this exercise we will use the following additional definitions:

```
def rep_image_family {A : Type}
  (F : Set (A → A)) (n : Nat) (B : Set A) : Set A :=
  match n with
  | 0 => B
  | k + 1 => {x : A | ∃ f ∈ F, x ∈ image f (rep_image_family F k B)}

def cumul_image_family {A : Type}
  (F : Set (A → A)) (B : Set A) : Set A :=
  {x : A | ∃ (n : Nat), x ∈ rep_image_family F n B}
```

The following theorems establish that if $F : \text{Set } (\text{A} \rightarrow \text{A})$ and $B : \text{Set } \text{A}$, then $\text{cumul_image_family}_F B$ is the closure of B under F . The first two are proven by `rfl`; the other two are for you to prove.

```
theorem rep_image_family_base {A : Type}
  (F : Set (A → A)) (B : Set A) : rep_image_family F 0 B = B := by rfl

theorem rep_image_family_step {A : Type}
  (F : Set (A → A)) (n : Nat) (B : Set A) :
  rep_image_family F (n + 1) B =
  {x : A | ∃ f ∈ F, x ∈ image f (rep_image_family F n B)} := by rfl

lemma rep_image_family_sub_closed {A : Type}
  (F : Set (A → A)) (B D : Set A)
  (h1 : B ⊆ D) (h2 : closed_family F D) :
  ∀ (n : Nat), rep_image_family F n B ⊆ D := sorry

theorem Exercise_6_5_3 {A : Type} (F : Set (A → A)) (B : Set A) :
  closure_family F B (cumul_image_family F B) := sorry
```

6.5. Closures Again

The next two exercises concern the following two definitions from Section 5.4:

```
def closed2 {A : Type} (f : A → A → A) (C : Set A) : Prop :=
  ∀ x ∈ C, ∀ y ∈ C, f x y ∈ C

def closure2 {A : Type} (f : A → A → A) (B C : Set A) : Prop :=
  smallestElt (sub A) C {D : Set A | B ⊆ D ∧ closed2 f D}
```

They also use the following definition, which extends the idea of the image of a set under a function to functions of two variables:

```
def image2 {A : Type} (f : A → A → A) (B : Set A) : Set A :=
  {z : A | ∃ (x y : A), x ∈ B ∧ y ∈ B ∧ z = f x y}
```

2. A natural way to try to find the closure of a set under a function of two variables would be to use the following definitions and theorems:

```
def rep_image2 {A : Type}
  (f : A → A → A) (n : Nat) (B : Set A) : Set A :=
  match n with
  | 0 => B
  | k + 1 => image2 f (rep_image2 f k B)

theorem rep_image2_base {A : Type} (f : A → A → A) (B : Set A) :
  rep_image2 f 0 B = B := by rfl

theorem rep_image2_step {A : Type}
  (f : A → A → A) (n : Nat) (B : Set A) :
  rep_image2 f (n + 1) B = image2 f (rep_image2 f n B) := by rfl

def cumul_image2 {A : Type} (f : A → A → A) (B : Set A) : Set A :=
  {x : A | ∃ (n : Nat), x ∈ rep_image2 f n B}
```

We could now try to prove that if $f : A \rightarrow A \rightarrow A$ and $B : \text{Set } A$, then $\text{cumul_image2 } f B$ is the closure of B under f . However, this approach doesn't work, because $\text{cumul_image2 } f B$ might not be closed under f .

Here is an incorrect informal argument that $\text{cumul_image2 } f B$ is closed under f . Suppose x and y are elements of $\text{cumul_image2 } f B$. This means that we can choose some natural number n such that $x \in \text{rep_image2 } f n B$ and $y \in \text{rep_image2 } f n B$. This implies that $f x y \in \text{image2 } f (\text{rep_image2 } f n B) = \text{rep_image2 } f (n + 1) B$, so $f x y \in \text{cumul_image2 } f B$.

Find the mistake in this informal argument by trying to turn it into a proof in Lean:

6.5. Closures Again

```
--You won't be able to complete this proof
theorem Exercise_6_5_6 {A : Type} (f : A → A → A) (B : Set A) :
  closed2 f (cumul_image2 f B) := sorry
```

3. In this exercise, we fix the mistake in the attempted proof in the previous exercise. Instead of repeatedly taking the image of a set, we repeatedly take the union of a set with its image:

```
def un_image2 {A : Type} (f : A → A → A) (B : Set A) : Set A :=
  B ∪ (image2 f B)

def rep_un_image2 {A : Type}
  (f : A → A → A) (n : Nat) (B : Set A) : Set A :=
  match n with
  | 0 => B
  | k + 1 => un_image2 f (rep_un_image2 f k B)

theorem rep_un_image2_base {A : Type} (f : A → A → A) (B : Set A) :
  rep_un_image2 f 0 B = B := by rfl

theorem rep_un_image2_step {A : Type}
  (f : A → A → A) (n : Nat) (B : Set A) :
  rep_un_image2 f (n + 1) B =
  un_image2 f (rep_un_image2 f n B) := by rfl

def cumul_un_image2 {A : Type}
  (f : A → A → A) (B : Set A) : Set A :=
  {x : A | ∃ (n : Nat), x ∈ rep_un_image2 f n B}
```

Now prove that if $f : A \rightarrow A \rightarrow A$ and $B : \text{Set } A$, then $\text{cumul_un_image2 } f B$ is the closure of B under f by completing the following proofs:

```
theorem Exercise_6_5_8a {A : Type} (f : A → A → A) (B : Set A) :
  ∀ (m n : Nat), m ≤ n →
  rep_un_image2 f m B ⊆ rep_un_image2 f n B := sorry

lemma rep_un_image2_sub_closed {A : Type} {f : A → A → A} {B D : Set A}
  (h1 : B ⊆ D) (h2 : closed2 f D) :
  ∀ (n : Nat), rep_un_image2 f n B ⊆ D := sorry

lemma closed_lemma
  {A : Type} {f : A → A → A} {B : Set A} {x y : A} {nx ny n : Nat}
```

6.5. Closures Again

```
(h1 : x ∈ rep_un_image2 f nx B) (h2 : y ∈ rep_un_image2 f ny B)
(h3 : nx ≤ n) (h4 : ny ≤ n) :
f x y ∈ cumul_un_image2 f B := sorry

theorem Exercise_6_5_8b {A : Type} (f : A → A → A) (B : Set A) :
closure2 f B (cumul_un_image2 f B) := sorry
```

The remaining exercises in this section use the following definitions:

```
def idExt (A : Type) : Set (A × A) := {(x, y) : A × A | x = y}

def rep_comp {A : Type} (R : Set (A × A)) (n : Nat) : Set (A × A) :=
match n with
| 0 => idExt A
| k + 1 => comp (rep_comp R k) R

def cumul_comp {A : Type} (R : Set (A × A)) : Set (A × A) :=
{(x, y) : A × A | ∃ n ≥ 1, (x, y) ∈ rep_comp R n}
```

4.

```
theorem rep_comp_one {A : Type} (R : Set (A × A)) :
rep_comp R 1 = R := sorry
```
5.

```
theorem Exercise_6_5_11 {A : Type} (R : Set (A × A)) :
∀ (m n : Nat), rep_comp R (m + n) =
comp (rep_comp R m) (rep_comp R n) := sorry
```
6.

```
lemma rep_comp_sub_trans {A : Type} {R S : Set (A × A)}
(h1 : R ⊆ S) (h2 : transitive (RelFromExt S)) :
∀ n ≥ 1, rep_comp R n ⊆ S := sorry
```
7.

```
theorem Exercise_6_5_14 {A : Type} (R : Set (A × A)) :
smallestElt (sub (A × A)) (cumul_comp R)
{S : Set (A × A) | R ⊆ S ∧ transitive (RelFromExt S)} := sorry
```

7 Number Theory

7.1. Greatest Common Divisors

The proofs in this chapter and the next are significantly longer than those in previous chapters. As a result, we will skip some details in the text, leaving proofs of a number of theorems as exercises for you. The most interesting of these exercises are included in the exercise lists at the ends of the sections; for the rest, you can compare your solutions to proofs that can be found in the Lean package that accompanies this book. Also, we will occasionally use theorems that we have not used before without explanation. If necessary, you can use `#check` to look up what they say.

Section 7.1 of *HTPI* introduces the Euclidean algorithm for computing the greatest common divisor (`gcd`) of two positive integers a and b . The motivation for the algorithm is the fact that if r is the remainder when a is divided by b , then any natural number that divides both a and b also divides r , and any natural number that divides both b and r also divides a .

Let's prove these statements in Lean. Recall that in Lean, the remainder when a is divided by b is called $a \bmod b$, and it is denoted $a \% b$. We'll prove the first statement, and leave the second as an exercise for you. It will be convenient for our work with greatest common divisors in Lean to let a and b be natural numbers rather than positive integers (thus allowing either of them to be zero).

```
theorem dvd_mod_of_dvd_a_b {a b d : Nat}
  (h1 : d | a) (h2 : d | b) : d | (a % b) := by
  set q : Nat := a / b
  have h3 : b * q + a % b = a := Nat.div_add_mod a b
  obtain (j : Nat) (h4 : a = d * j) from h1
  obtain (k : Nat) (h5 : b = d * k) from h2
  define --Goal : ∃ (c : Nat), a % b = d * c
  apply Exists.intro (j - k * q)
  show a % b = d * (j - k * q) from
    calc a % b
      _ = b * q + a % b - b * q := (Nat.add_sub_cancel_left _ _).symm
      _ = a - b * q := by rw [h3]
      _ = d * j - d * (k * q) := by rw [h4, h5, mul_assoc]
      _ = d * (j - k * q) := (Nat.mul_sub_left_distrib _ _).symm
```

```
done

theorem dvd_a_of_dvd_b_mod {a b d : Nat}
  (h1 : d | b) (h2 : d | (a % b)) : d | a := sorry
```

These theorems tell us that the gcd of a and b is the same as the gcd of b and $a \% b$, which suggests that the following recursive definition should compute the gcd of a and b :

```
def gcd (a b : Nat) : Nat :=
  match b with
  | 0 => a
  | n + 1 => gcd (n + 1) (a % (n + 1))
```

Unfortunately, Lean puts a red squiggle under `gcd`, and it displays in the Infoview a long error message that begins `fail to show termination`. What is Lean complaining about?

The problem is that recursive definitions are dangerous. To understand the danger, consider the following recursive definition:

```
def loop (n : Nat) : Nat := loop (n + 1)
```

Suppose we try to use this definition to compute `loop 3`. The definition would lead us to perform the following calculation:

```
loop 3 = loop 4 = loop 5 = loop 6 = ...
```

Clearly this calculation will go on forever and will never produce an answer. So the definition of `loop` does not actually succeed in defining a function from `Nat` to `Nat`.

Lean insists that recursive definitions must avoid such nonterminating calculations. Why did it accept all of our previous recursive definitions? The reason is that in each case, the definition of the value of the function at a natural number n referred only to values of the function at numbers smaller than n . Since a decreasing list of natural numbers cannot go on forever, such definitions lead to calculations that are guaranteed to terminate.

What about our recursive definition of `gcd a b`? This function has two arguments, a and b , and when $b = n + 1$, the definition asks us to compute `gcd (n + 1) (a % (n + 1))`. The first argument here could actually be larger than the first argument in the value we are trying to compute, `gcd a b`. But the second argument will always be smaller, and that will suffice to guarantee that the calculation terminates. We can tell Lean to focus on the second argument b by adding a `termination_by` clause to the end of our recursive definition:

```
def gcd (a b : Nat) : Nat :=
  match b with
  | 0 => a
  | n + 1 => gcd (n + 1) (a % (n + 1))
termination_by b
```

Unfortunately, Lean still isn't satisfied, but the error message this time is more helpful. The message says that Lean failed to prove termination, and at the end of the message it says that the goal it failed to prove is $a \% (n + 1) < n + 1$, which is precisely what is needed to show that the second argument of $\text{gcd } (n + 1) (a \% (n + 1))$ is smaller than the second argument of $\text{gcd } a b$ when $b = n + 1$. We'll need to provide a proof of this goal to convince Lean to accept our recursive definition. Fortunately, it's not hard to prove:

```
lemma mod_succ_lt (a n : Nat) : a \% (n + 1) < n + 1 := by
  have h : n + 1 > 0 := Nat.succ_pos n
  show a \% (n + 1) < n + 1 from Nat.mod_lt a h
  done
```

Lean's error message suggests several ways to fix the problem with our recursive definition. We'll use the first suggestion: Use 'have'-expressions to prove the remaining goals. Here, finally, is the definition of gcd that Lean is willing to accept. (You can ignore the initial line `@[semireducible]`. For technical reasons that we won't go into, this line is needed to make this complicated recursive definition work in proofs the same way that our previous, simpler recursive definitions did.)

```
@[semireducible]
def gcd (a b : Nat) : Nat :=
  match b with
  | 0 => a
  | n + 1 =>
    have : a \% (n + 1) < n + 1 := mod_succ_lt a n
    gcd (n + 1) (a \% (n + 1))
  termination_by b
```

Notice that in the `have` expression, we have not bothered to specify an identifier for the assertion being proven, since we never need to refer to it. Let's try out our gcd function:

```
#eval gcd 672 161 --Answer: 7. Note 672 = 7 * 96 and 161 = 7 * 23.
```

To establish the main properties of $\text{gcd } a b$ we'll need several lemmas. We prove some of them and leave others as exercises.

```

lemma gcd_base (a : Nat) : gcd a 0 = a := rfl

lemma gcd_nonzero (a : Nat) {b : Nat} (h : b ≠ 0) :
  gcd a b = gcd b (a % b) := by
  obtain (n : Nat) (h2 : b = n + 1) from exists_eq_add_one_of_ne_zero h
  rewrite [h2] --Goal : gcd a (n + 1) = gcd (n + 1) (a % (n + 1))
  rfl
  done

lemma mod_nonzero_lt (a : Nat) {b : Nat} (h : b ≠ 0) : a % b < b := sorry

lemma dvd_self (n : Nat) : n | n := sorry

```

One of the most important properties of $\text{gcd } a \ b$ is that it divides both a and b . We prove it by strong induction on b .

```

theorem gcd_dvd : ∀ (b a : Nat), (gcd a b) | a ∧ (gcd a b) | b := by
  by_strong_induc
  fix b : Nat
  assume ih : ∀ b_1 < b, ∀ (a : Nat), (gcd a b_1) | a ∧ (gcd a b_1) | b_1
  fix a : Nat
  by_cases h1 : b = 0
  · -- Case 1. h1 : b = 0
    rewrite [h1, gcd_base] --Goal: a | a ∧ a | 0
    apply And.intro (dvd_self a)
    define
    apply Exists.intro 0
    rfl
    done
  · -- Case 2. h1 : b ≠ 0
    rewrite [gcd_nonzero a h1]
    --Goal : gcd b (a % b) | a ∧ gcd b (a % b) | b
    have h2 : a % b < b := mod_nonzero_lt a h1
    have h3 : (gcd b (a % b)) | b ∧ (gcd b (a % b)) | (a % b) :=
      ih (a % b) h2 b
    apply And.intro _ h3.left
    show (gcd b (a % b)) | a from dvd_a_of_dvd_b_mod h3.left h3.right
    done
  done

```

You may wonder why we didn't start the proof like this:

7.1. Greatest Common Divisors

```
theorem gcd_dvd : ∀ (a b : Nat), (gcd a b) | a ∧ (gcd a b) | b := by
  fix a : Nat
  by_strong_induc
  fix b : Nat
  assume ih : ∀ b_1 < b, (gcd a b_1) | a ∧ (gcd a b_1) | b_1
```

In fact, this approach wouldn't have worked. It is an interesting exercise to try to complete this version of the proof and see why it fails.

Another interesting question is why we asserted both $(\text{gcd } a \ b) \mid a$ and $(\text{gcd } a \ b) \mid b$ in the same theorem. Wouldn't it have been easier to give separate proofs of the statements $\forall (b \ a : \text{Nat})$, $(\text{gcd } a \ b) \mid a$ and $\forall (b \ a : \text{Nat})$, $(\text{gcd } a \ b) \mid b$? Again, you might find it enlightening to see why that wouldn't have worked. However, now that we have proven both divisibility statements, we can state them as separate theorems:

```
theorem gcd_dvd_left (a b : Nat) : (gcd a b) | a := (gcd_dvd b a).left
theorem gcd_dvd_right (a b : Nat) : (gcd a b) | b := (gcd_dvd b a).right
```

Next we turn to Theorem 7.1.4 in *HTPI*, which says that there are integers s and t such that $\text{gcd}(a, b) = sa + tb$. (We say that $\text{gcd}(a, b)$ can be written as a *linear combination* of a and b .) In *HTPI*, this is proven by using an extended version of the Euclidean algorithm to compute the coefficients s and t . Here we will use a different recursive procedure to compute s and t . If $b = 0$, then $\text{gcd}(a, b) = a = 1 \cdot a + 0 \cdot b$, so we can use the values $s = 1$ and $t = 0$. Otherwise, let q and r be the quotient and remainder when a is divided by b . Then $a = bq + r$ and $\text{gcd}(a, b) = \text{gcd}(b, r)$. Now suppose that we have already computed integers s' and t' such that

$$\text{gcd}(b, r) = s'b + t'r.$$

Then

$$\begin{aligned} \text{gcd}(a, b) &= \text{gcd}(b, r) = s'b + t'r \\ &= s'b + t'(a - bq) = t'a + (s' - t'q)b. \end{aligned}$$

Thus, to write $\text{gcd}(a, b) = sa + tb$ we can use the values

$$s = t', \quad t = s' - t'q. \tag{*}$$

We will use these equations as the basis for recursive definitions of Lean functions `gcd_c1` and `gcd_c2` such that the required coefficients can be obtained from the formulas `s = gcd_c1 a b` and `t = gcd_c2 a b`. Notice that `s` and `t` could be negative, so they must have type `Int`, not `Nat`. As a result, in definitions and theorems involving `gcd_c1` and `gcd_c2` we will sometimes have to deal with coercion of natural numbers to integers.

7.1. Greatest Common Divisors

The functions `gcd_c1` and `gcd_c2` will be *mutually recursive*; in other words, each will be defined not only in terms of itself but also in terms of the other. Fortunately, Lean allows for such mutual recursion. Here are the definitions we will use.

```
mutual
  @[semireducible]
  def gcd_c1 (a b : Nat) : Int :=
    match b with
    | 0 => 1
    | n + 1 =>
      have : a % (n + 1) < n + 1 := mod_succ_lt a n
      gcd_c2 (n + 1) (a % (n + 1))
      --Corresponds to s = t'
    termination_by b

  @[semireducible]
  def gcd_c2 (a b : Nat) : Int :=
    match b with
    | 0 => 0
    | n + 1 =>
      have : a % (n + 1) < n + 1 := mod_succ_lt a n
      gcd_c1 (n + 1) (a % (n + 1)) -
      (gcd_c2 (n + 1) (a % (n + 1))) * ↑(a / (n + 1))
      --Corresponds to t = s' - t'q
    termination_by b
end
```

Notice that in the definition of `gcd_c2`, the quotient $a / (n + 1)$ is computed using natural-number division, but it is then coerced to be an integer so that it can be multiplied by the integer `gcd_c2 (n + 1) (a % (n + 1))`.

Our main theorem about these functions is that they give the coefficients needed to write $\text{gcd } a \ b$ as a linear combination of a and b . As usual, stating a few lemmas first helps with the proof. We leave the proofs of two of them as exercises for you (hint: imitate the proof of `gcd_nonzero` above).

```
lemma gcd_c1_base (a : Nat) : gcd_c1 a 0 = 1 := rfl

lemma gcd_c1_nonzero (a : Nat) {b : Nat} (h : b ≠ 0) :
  gcd_c1 a b = gcd_c2 b (a % b) := sorry

lemma gcd_c2_base (a : Nat) : gcd_c2 a 0 = 0 := rfl
```

```
lemma gcd_c2_nonzero (a : Nat) {b : Nat} (h : b ≠ 0) :
gcd_c2 a b = gcd_c1 b (a % b) - (gcd_c2 b (a % b)) * ↑(a / b) := sorry
```

With that preparation, we are ready to prove that $\text{gcd_c1 } a \ b$ and $\text{gcd_c2 } a \ b$ give coefficients for expressing $\text{gcd } a \ b$ as a linear combination of a and b . Of course, the theorem is proven by strong induction. For clarity, we'll write the coercions explicitly in this proof. We'll make a few comments after the proof that may help you follow the details.

```
theorem gcd_lin_comb : ∀ (b a : Nat),
  (gcd_c1 a b) * ↑a + (gcd_c2 a b) * ↑b = ↑(gcd a b) := by
  by_strong_induc
  fix b : Nat
  assume ih : ∀ b_1 < b, ∀ (a : Nat),
    (gcd_c1 a b_1) * ↑a + (gcd_c2 a b_1) * ↑b_1 = ↑(gcd a b_1)
  fix a : Nat
  by_cases h1 : b = 0
  . -- Case 1. h1 : b = 0
    rewrite [h1, gcd_c1_base, gcd_c2_base, gcd_base]
    --Goal : 1 * ↑a + 0 * ↑0 = ↑a
    ring
    done
  . -- Case 2. h1 : b ≠ 0
    rewrite [gcd_c1_nonzero a h1, gcd_c2_nonzero a h1, gcd_nonzero a h1]
    --Goal : gcd_c2 b (a % b) * ↑a +
    -- (gcd_c1 b (a % b) - gcd_c2 b (a % b)) * ↑(a / b) * ↑b =
    -- ↑(gcd b (a % b))
    set r : Nat := a % b
    set q : Nat := a / b
    set s : Int := gcd_c1 b r
    set t : Int := gcd_c2 b r
    --Goal : t * ↑a + (s - t * ↑q) * ↑b = ↑(gcd b r)
    have h2 : r < b := mod_nonzero_lt a h1
    have h3 : s * ↑b + t * ↑r = ↑(gcd b r) := ih r h2 b
    have h4 : b * q + r = a := Nat.div_add_mod a b
    rewrite [←h3, ←h4]
    rewrite [Nat.cast_add, Nat.cast_mul]
    --Goal : t * (↑b * ↑q + ↑r) + (s - t * ↑q) * ↑b = s * ↑b + t * ↑r
    ring
    done
  done
```

In case 2, we have introduced the variables r , q , s , and t to simplify the notation. Notice that

7.1. Greatest Common Divisors

the `set` tactic automatically plugs in this notation in the goal. After the step `rewrite [←h3, ←h4]`, the goal contains the expression $\uparrow(b * q + r)$. You can use the `#check` command to see why `Nat.cast_add` and `Nat.cast_mul` convert this expression to first $\uparrow(b * q) + \uparrow r$ and then $\uparrow b * \uparrow q + \uparrow r$. Without those steps, the `ring` tactic would not have been able to complete the proof.

We can try out the functions `gcd_c1` and `gcd_c2` as follows:

```
#eval gcd_c1 672 161 --Answer: 6
#eval gcd_c2 672 161 --Answer: -25
--Note 6 * 672 - 25 * 161 = 4032 - 4025 = 7 = gcd 672 161
```

Finally, we turn to Theorem 7.1.6 in *HTPI*, which expresses one of the senses in which $\gcd a b$ is the *greatest* common divisor of a and b . Our proof follows the strategy of the proof in *HTPI*, with one additional step: we begin by using the theorem `Int.natCast_dvd_natCast` to change the goal from $d \mid \gcd a b$ to $\uparrow d \mid \uparrow(\gcd a b)$ (where the coercions are from `Nat` to `Int`), so that the rest of the proof can work with integer algebra rather than natural-number algebra.

```
theorem Theorem_7_1_6 {d a b : Nat} (h1 : d ∣ a) (h2 : d ∣ b) :
  d ∣ gcd a b := by
  rewrite [←Int.natCast_dvd_natCast]      --Goal : ↑d ∣ ↑(gcd a b)
  set s : Int := gcd_c1 a b
  set t : Int := gcd_c2 a b
  have h3 : s * ↑a + t * ↑b = ↑(gcd a b) := gcd_lin_comb b a
  rewrite [←h3]                      --Goal : ↑d ∣ s * ↑a + t * ↑b
  obtain (j : Nat) (h4 : a = d * j) from h1
  obtain (k : Nat) (h5 : b = d * k) from h2
  rewrite [h4, h5, Nat.cast_mul, Nat.cast_mul]
  --Goal : ↑d ∣ s * (↑d * ↑j) + t * (↑d * ↑k)
  define
  apply Exists.intro (s * ↑j + t * ↑k)
  ring
  done
```

We will ask you in the exercises to prove that, among the common divisors of a and b , $\gcd a b$ is the greatest with respect to the usual ordering of the natural numbers (as long as $\gcd a b \neq 0$).

Exercises

1. `theorem dvd_a_of_dvd_b_mod {a b d : Nat} (h1 : d | b) (h2 : d | (a % b)) : d | a := sorry`
2. `lemma gcd_comm_lt {a b : Nat} (h : a < b) : gcd a b = gcd b a := sorry`
`theorem gcd_comm (a b : Nat) : gcd a b = gcd b a := sorry`
3. `theorem Exercise_7_1_5 (a b : Nat) (n : Int) : (exists (s t : Int), s * a + t * b = n) ↔ (↑(gcd a b) : Int) | n := sorry`
4. `theorem Exercise_7_1_6 (a b c : Nat) : gcd a b = gcd (a + b * c) b := sorry`
5. `theorem gcd_is_nonzero {a b : Nat} (h : a ≠ 0 ∨ b ≠ 0) : gcd a b ≠ 0 := sorry`
6. `theorem gcd_greatest {a b d : Nat} (h1 : gcd a b ≠ 0) (h2 : d | a) (h3 : d | b) : d ≤ gcd a b := sorry`
7. `lemma Lemma_7_1_10a {a b : Nat} (n : Nat) (h : a | b) : (n * a) | (n * b) := sorry`
`lemma Lemma_7_1_10b {a b n : Nat} (h1 : n ≠ 0) (h2 : (n * a) | (n * b)) : a | b := sorry`
`lemma Lemma_7_1_10c {a b : Nat} (h1 : a | b) (h2 : b | a) : a = b := sorry`
`theorem Exercise_7_1_10 (a b n : Nat) : gcd (n * a) (n * b) = n * gcd a b := sorry`

7.2. Prime Factorization

A natural number n is said to be *prime* if it is at least 2 and it cannot be written as a product of two smaller natural numbers. Of course, we can write this definition in Lean.

```
def prime (n : Nat) : Prop :=
  2 ≤ n ∧ ¬∃ (a b : Nat), a * b = n ∧ a < n ∧ b < n
```

The main goal of Section 7.2 of *HTPI* is to prove that every positive integer has a unique prime factorization; that is, it can be written in a unique way as the product of a nondecreasing list of prime numbers. To get started on this goal, we first prove that every number greater than or equal to 2 has a prime factor. We leave one lemma as an exercise for you (it is a natural-number version of `Theorem_3_3_7`).

```
def prime_factor (p n : Nat) : Prop := prime p ∧ p ∣ n

lemma dvd_trans {a b c : Nat} (h1 : a ∣ b) (h2 : b ∣ c) : a ∣ c := sorry

lemma exists_prime_factor : ∀ (n : Nat), 2 ≤ n →
  ∃ (p : Nat), prime_factor p n := by
  by_strong_induc
  fix n : Nat
  assume ih : ∀ n_1 < n, 2 ≤ n_1 → ∃ (p : Nat), prime_factor p n_1
  assume h1 : 2 ≤ n
  by_cases h2 : prime n
  · -- Case 1. h2 : prime n
    apply Exists.intro n
    define          --Goal : prime n ∧ n ∣ n
    show prime n ∧ n ∣ n from And.intro h2 (dvd_self n)
    done
  · -- Case 2. h2 : ¬prime n
    define at h2
    --h2 : ¬(2 ≤ n ∧ ∃ (a b : Nat), a * b = n ∧ a < n ∧ b < n)
    demorgan at h2
    disj_syll h2 h1
    obtain (a : Nat) (h3 : ∃ (b : Nat), a * b = n ∧ a < n ∧ b < n) from h2
    obtain (b : Nat) (h4 : a * b = n ∧ a < n ∧ b < n) from h3
    have h5 : 2 ≤ a := by
      by_contra h6
      have h7 : a ≤ 1 := by linarith
      have h8 : n ≤ b :=
        calc n
        _ = a * b := h4.left.symm
        _ ≤ 1 * b := by rel [h7]
        _ = b := by ring
      linarith      --n ≤ b contradicts b < n
      done
    have h6 : ∃ (p : Nat), prime_factor p a := ih a h4.right.left h5
    obtain (p : Nat) (h7 : prime_factor p a) from h6
    apply Exists.intro p
    define          --Goal : prime p ∧ p ∣ n
```

```

define at h7      --h7 : prime p ∧ p | a
apply And.intro h7.left
have h8 : a | n := by
  apply Exists.intro b
  show n = a * b from (h4.left).symm
  done
show p | n from dvd_trans h7.right h8
done
done

```

Of course, by the well-ordering principle, an immediate consequence of this lemma is that every number greater than or equal to 2 has a *smallest* prime factor.

```

lemma exists_least_prime_factor {n : Nat} (h : 2 ≤ n) :
  ∃ (p : Nat), prime_factor p n ∧
  ∀ (q : Nat), prime_factor q n → p ≤ q := by
set S : Set Nat := {p : Nat | prime_factor p n}
have h2 : ∃ (p : Nat), p ∈ S := exists_prime_factor n h
show ∃ (p : Nat), prime_factor p n ∧
  ∀ (q : Nat), prime_factor q n → p ≤ q from well_ord_princ S h2
done

```

To talk about prime factorizations of positive integers, we'll need to introduce a new type. If U is any type, then `List U` is the type of lists of objects of type U . Such a list is written in square brackets, with the entries separated by commas. For example, `[3, 7, 1]` has type `List Nat`. The notation `[]` denotes the empty list, and if a has type U and l has type `List U`, then $a :: l$ denotes the list consisting of a followed by the entries of l . The empty list is sometimes called the `nil` list, and the operation of constructing a list $a :: l$ from a and l is called `cons` (short for *construct*). Every list can be constructed by applying the `cons` operation repeatedly, starting with the `nil` list. For example,

$$[3, 7, 1] = 3 :: [7, 1] = 3 :: (7 :: [1]) = 3 :: (7 :: (1 :: [])).$$

If l has type `List U` and a has type U , then $a \in l$ means that a is one of the entries in the list l . For example, $7 \in [3, 7, 1]$. Lean knows several theorems about this notation:

```

@List.not_mem_nil : ∀ {α : Type u_1} {a : α},
  a ∉ []

@List.mem_cons : ∀ {α : Type u_1} {b : α} {l : List α} {a : α},
  a ∈ b :: l ↔ a = b ∨ a ∈ l

@List.mem_cons_self : ∀ {α : Type u_1} {a : α} {l : List α},
  a ∈ a :: l

```

```
a ∈ a :: l
```

```
@List.mem_cons_of_mem : ∀ {α : Type u_1} (y : α) {a : α} {l : List α},
a ∈ l → a ∈ y :: l
```

The first two theorems give the conditions under which something is a member of the `nil` list or a list constructed by `cons`, and the last two are easy consequences of the second.

To define prime factorizations, we must define several concepts first. Some of these concepts are most easily defined recursively.

```
def all_prime (l : List Nat) : Prop := ∀ p ∈ l, prime p

def nondec (l : List Nat) : Prop :=
match l with
| [] => True    --Of course, True is a proposition that is always true
| n :: L => (∀ m ∈ L, n ≤ m) ∧ nondec L

def nondec_prime_list (l : List Nat) : Prop := all_prime l ∧ nondec l

def prod (l : List Nat) : Nat :=
match l with
| [] => 1
| n :: L => n * (prod L)

def prime_factorization (n : Nat) (l : List Nat) : Prop :=
nondec_prime_list l ∧ prod l = n
```

According to these definitions, `all_prime l` means that every member of the list `l` is prime, `nondec l` means that every member of `l` is less than or equal to all later members, `prod l` is the product of all members of `l`, and `prime_factorization n l` means that `l` is a nondecreasing list of prime numbers whose product is `n`. It will be convenient to spell out some consequences of these definitions in several lemmas:

```
lemma all_prime_nil : all_prime [] := by
  define      --Goal : ∀ p ∈ [], prime p
  fix p : Nat
  contrapos --Goal : ¬prime p → p ∉ []
  assume h1 : ¬prime p
  show p ∉ [] from List.not_mem_nil
  done

lemma all_prime_cons (n : Nat) (L : List Nat) :
```

```

all_prime (n :: L) ↔ prime n ∧ all_prime L := by
apply Iff.intro
· -- (→)
  assume h1 : all_prime (n :: L) --Goal : prime n ∧ all_prime L
  define at h1 --h1 : ∀ p ∈ n :: L, prime p
  apply And.intro (h1 n List.mem_cons_self)
  define --Goal : ∀ p ∈ L, prime p
  fix p : Nat
  assume h2 : p ∈ L
  show prime p from h1 p (List.mem_cons_of_mem n h2)
  done
· -- (←)
  assume h1 : prime n ∧ all_prime L --Goal : all_prime (n :: l)
  define : all_prime L at h1
  define
  fix p : Nat
  assume h2 : p ∈ n :: L
  rewrite [List.mem_cons] at h2 --h2 : p = n ∨ p ∈ L
  by_cases on h2
  · -- Case 1. h2 : p = n
    rewrite [h2]
    show prime n from h1.left
    done
  · -- Case 2. h2 : p ∈ L
    show prime p from h1.right p h2
    done
  done
done

lemma nondec_nil : nondec [] := by
  define --Goal : True
  trivial --trivial proves some obviously true statements, such as True
done

lemma nondec_cons (n : Nat) (L : List Nat) :
  nondec (n :: L) ↔ (∀ m ∈ L, n ≤ m) ∧ nondec L := by rfl

lemma prod_nil : prod [] = 1 := by rfl

lemma prod_cons : prod (n :: L) = n * (prod L) := by rfl

```

Before we can prove the existence of prime factorizations, we will need one more fact: every

7.2. Prime Factorization

member of a list of natural numbers divides the product of the list. The proof will be by induction on the length of the list, so we will need to know how to work with lengths of lists in Lean. If l is a list, then the length of l is `List.length l`, which can also be written more briefly as `l.length`. We'll need a few more theorems about lists:

```
@List.length_eq_zero_iff : ∀ {α : Type u_1} {l : List α},
l.length = 0 ↔ l = []

@List.length_cons : ∀ {α : Type u_1} {a : α} {as : List α},
(a :: as).length = as.length + 1

@List.exists_cons_of_ne_nil : ∀ {α : Type u_1} {l : List α},
l ≠ [] → ∃ (b : α) (l' : List α), l = b :: l'
```

And we'll need one more lemma, which follows from the three theorems above; we leave the proof as an exercise for you:

```
lemma exists_cons_of_length_eq_succ {A : Type}
{l : List A} {n : Nat} (h : l.length = n + 1) :
∃ (a : A) (L : List A), l = a :: L ∧ L.length = n := sorry
```

We can now prove that every member of a list of natural numbers divides the product of the list. After proving it by induction on the length of the list, we restate the lemma in a more convenient form.

```
lemma list_elt_dvd_prod_by_length (a : Nat) : ∀ (n : Nat),
  ∀ (l : List Nat), l.length = n → a ∈ l → a | prod l := by
  by_induc
  · --Base Case
    fix l : List Nat
    assume h1 : l.length = 0
    rewrite [List.length_eq_zero_iff] at h1      --h1 : l = []
    rewrite [h1]                                     --Goal : a ∈ [] → a | prod []
    contrapos
    assume h2 : ¬a | prod []
    show a ∉ [] from List.not_mem_nil
    done
  · -- Induction Step
    fix n : Nat
    assume ih : ∀ (l : List Nat), l.length = n → a ∈ l → a | prod l
    fix l : List Nat
    assume h1 : l.length = n + 1                  --Goal : a ∈ l → a | prod l
    obtain (b : Nat) (h2 : ∃ (L : List Nat),
```

```

l = b :: L ∧ L.length = n) from exists_cons_of_length_eq_succ h1
obtain (L : List Nat) (h3 : l = b :: L ∧ L.length = n) from h2
have h4 : a ∈ L → a | prod L := ih L h3.right
assume h5 : a ∈ l
rewrite [h3.left, prod_cons]           --Goal : a | b * prod L
rewrite [h3.left, List.mem_cons] at h5 --h5 : a = b ∨ a ∈ L
by_cases on h5
· -- Case 1. h5 : a = b
  apply Exists.intro (prod L)
  rewrite [h5]
  rfl
  done
· -- Case 2. h5 : a ∈ L
  have h6 : a | prod L := h4 h5
  have h7 : prod L | b * prod L := by
    apply Exists.intro b
    ring
    done
  show a | b * prod L from dvd_trans h6 h7
  done
done
done

lemma list_elt_dvd_prod {a : Nat} {l : List Nat}
  (h : a ∈ l) : a | prod l := by
set n : Nat := l.length
have h1 : l.length = n := by rfl
show a | prod l from list_elt_dvd_prod_by_length a n l h1 h
done

```

The proof that every positive integer has a prime factorization is now long but straightforward.

```

lemma exists_prime_factorization : ∀ (n : Nat), n ≥ 1 →
  ∃ (l : List Nat), prime_factorization n l := by
  by_strong_induc
  fix n : Nat
  assume ih : ∀ n_1 < n, n_1 ≥ 1 →
    ∃ (l : List Nat), prime_factorization n_1 l
  assume h1 : n ≥ 1
  by_cases h2 : n = 1
  · -- Case 1. h2 : n = 1

```

```

apply Exists.intro []
define
apply And.intro
· -- Proof of nondec_prime_list []
define
show all_prime [] ∧ nondec [] from
And.intro all_prime_nil nondec_nil
done
· -- Proof of prod [] = n
rewrite [prod_nil, h2]
rfl
done
done
· -- Case 2. h2 : n ≠ 1
have h3 : n ≥ 2 := lt_of_le_of_ne' h1 h2
obtain (p : Nat) (h4 : prime_factor p n ∧ ∀ (q : Nat),
prime_factor q n → p ≤ q) from exists_least_prime_factor h3
have p_prime_factor : prime_factor p n := h4.left
define at p_prime_factor
have p_prime : prime p := p_prime_factor.left
have p_dvd_n : p ∣ n := p_prime_factor.right
have p_least : ∀ (q : Nat), prime_factor q n → p ≤ q := h4.right
obtain (m : Nat) (n_eq_pm : n = p * m) from p_dvd_n
have h5 : m ≠ 0 := by
contradict h1 with h6
have h7 : n = 0 :=
calc n
_ = p * m := n_eq_pm
_ = p * 0 := by rw [h6]
_ = 0 := by ring
rewrite [h7]
decide
done
have m_pos : 0 < m := Nat.pos_of_ne_zero h5
have m_lt_n : m < n := by
define at p_prime
show m < n from
calc m
_ < m + m := by linarith
_ = 2 * m := by ring
_ ≤ p * m := by rel [p_prime.left]
_ = n := n_eq_pm.symm

```

```

done
obtain (L : List Nat) (h6 : prime_factorization m L)
  from ih m m_lt_n m_pos
define at h6
have ndpl_L : nondec_prime_list L := h6.left
define at ndpl_L
apply Exists.intro (p :: L)
define
apply And.intro
• -- Proof of nondec_prime_list (p :: L)
define
apply And.intro
• -- Proof of all_prime (p :: L)
rewrite [all_prime_cons]
show prime p ∧ all_prime L from And.intro p_prime ndpl_L.left
done
• -- Proof of nondec (p :: L)
rewrite [nondec_cons]
apply And.intro _ ndpl_L.right
fix q : Nat
assume q_in_L : q ∈ L
have h7 : q ∣ prod L := list_elt_dvd_prod q_in_L
rewrite [h6.right] at h7 --h7 : q ∣ m
have h8 : m ∣ n := by
  apply Exists.intro p
  rewrite [n_eq_pm]
  ring
  done
have q_dvd_n : q ∣ n := dvd_trans h7 h8
have ap_L : all_prime L := ndpl_L.left
define at ap_L
have q_prime_factor : prime_factor q n :=
  And.intro (ap_L q q_in_L) q_dvd_n
show p ≤ q from p_least q q_prime_factor
done
done
• -- Proof of prod (p :: L) = n
rewrite [prod_cons, h6.right, n_eq_pm]
rfl
done
done
done

```

We now turn to the proof that the prime factorization of a positive integer is unique. In preparation for that proof, *HTPI* defines two numbers to be *relatively prime* if their greatest common divisor is 1, and then it uses that concept to prove two theorems, 7.2.2 and 7.2.3. Here are similar proofs of those theorems in Lean, with the proof of one lemma left as an exercise. In the proof of Theorem 7.2.2, we begin, as we did in the proof of Theorem 7.1.6, by converting the goal from natural numbers to integers so that we can use integer algebra.

```

def rel_prime (a b : Nat) : Prop := gcd a b = 1

theorem Theorem_7_2_2 {a b c : Nat}
  (h1 : c | a * b) (h2 : rel_prime a c) : c | b := by
  rewrite [←Int.natCast_dvd_natCast] --Goal : ↑c | ↑b
  define at h1; define at h2; define
  obtain (j : Nat) (h3 : a * b = c * j) from h1
  set s : Int := gcd_c1 a c
  set t : Int := gcd_c2 a c
  have h4 : s * ↑a + t * ↑c = ↑(gcd a c) := gcd_lin_comb c a
  rewrite [h2, Nat.cast_one] at h4 --h4 : s * ↑a + t * ↑c = (1 : Int)
  apply Exists.intro (s * ↑j + t * ↑b)
  show ↑b = ↑c * (s * ↑j + t * ↑b) from
  calc ↑b
    _ = (1 : Int) * ↑b := (one_mul _).symm
    _ = (s * ↑a + t * ↑c) * ↑b := by rw [h4]
    _ = s * (↑a * ↑b) + t * ↑c * ↑b := by ring
    _ = s * (↑c * ↑j) + t * ↑c * ↑b := by
      rw [←Nat.cast_mul a b, h3, Nat.cast_mul c j]
    _ = ↑c * (s * ↑j + t * ↑b) := by ring
  done

lemma dvd_prime {a p : Nat}
  (h1 : prime p) (h2 : a | p) : a = 1 ∨ a = p := sorry

lemma rel_prime_of_prime_not_dvd {a p : Nat}
  (h1 : prime p) (h2 : ¬p | a) : rel_prime a p := by
  have h3 : gcd a p | a := gcd_dvd_left a p
  have h4 : gcd a p | p := gcd_dvd_right a p
  have h5 : gcd a p = 1 ∨ gcd a p = p := dvd_prime h1 h4
  have h6 : gcd a p ≠ p := by
    contradict h2 with h6
  rewrite [h6] at h3
  show p | a from h3
  done
disj_syll h5 h6

```

```

show rel_prime a p from h5
done

theorem Theorem_7_2_3 {a b p : Nat}
  (h1 : prime p) (h2 : p ∣ a * b) : p ∣ a ∨ p ∣ b := by
  or_right with h3
  have h4 : rel_prime a p := rel_prime_of_prime_not_dvd h1 h3
  show p ∣ b from Theorem_7_2_2 h2 h4
  done

```

Theorem 7.2.4 in *HTPI* extends Theorem 7.2.3 to show that if a prime number divides the product of a list of natural numbers, then it divides one of the numbers in the list. (Theorem 7.2.3 is the case of a list of length two.) The proof in *HTPI* is by induction on the length of the list, and we could use that method to prove the theorem in Lean. But look back at our proof of the lemma `list_elt_dvd_prod_by_length`, which also used induction on the length of a list. In the base case, we ended up proving that the `nil` list has the property stated in the lemma, and in the induction step we proved that if a list `L` has the property, then so does any list of the form `b :: L`. We could think of this as a kind of “induction on lists.” As we observed earlier, every list can be constructed by starting with the `nil` list and applying `cons` finitely many times. It follows that if the `nil` list has some property, and applying the `cons` operation to a list with the property produces another list with the property, then all lists have the property. (In fact, a similar principle was at work in our recursive definitions of `nondec l` and `prod l`.)

Lean has a theorem called `List.rec` that can be used to justify induction on lists. This is a little more convenient than induction on the length of a list, so we’ll use it to prove Theorem 7.2.4. The proof uses two lemmas, whose proofs we leave as exercises for you.

```

lemma eq_one_of_dvd_one {n : Nat} (h : n ∣ 1) : n = 1 := sorry

lemma prime_not_one {p : Nat} (h : prime p) : p ≠ 1 := sorry

theorem Theorem_7_2_4 {p : Nat} (h1 : prime p) :
  ∀ (l : List Nat), p ∣ prod l → ∃ a ∈ l, p ∣ a := by
  apply List.rec
  · -- Base Case. Goal : p ∣ prod [] → ∃ a ∈ [], p ∣ a
    rewrite [prod_nil]
    assume h2 : p ∣ 1
    show ∃ a ∈ [], p ∣ a from
      absurd (eq_one_of_dvd_one h2) (prime_not_one h1)
    done
  · -- Induction Step

```

```

fix b : Nat
fix L : List Nat
assume ih : p | prod L → ∃ a ∈ L, p | a
--Goal : p | prod (b :: L) → ∃ a ∈ b :: L, p | a
assume h2 : p | prod (b :: L)
rewrite [prod_cons] at h2
have h3 : p | b ∨ p | prod L := Theorem_7_2_3 h1 h2
by_cases on h3
· -- Case 1. h3 : p | b
  apply Exists.intro b
  show b ∈ b :: L ∧ p | b from
    And.intro List.mem_cons_self h3
  done
· -- Case 2. h3 : p | prod L
  obtain (a : Nat) (h4 : a ∈ L ∧ p | a) from ih h3
  apply Exists.intro a
  show a ∈ b :: L ∧ p | a from
    And.intro (List.mem_cons_of_mem b h4.left) h4.right
  done
done
done

```

In Theorem 7.2.4, if all members of the list l are prime, then we can conclude not merely that p divides some member of l , but that p is one of the members.

```

lemma prime_in_list {p : Nat} {l : List Nat}
  (h1 : prime p) (h2 : all_prime l) (h3 : p | prod l) : p ∈ l := by
  obtain (a : Nat) (h4 : a ∈ l ∧ p | a) from Theorem_7_2_4 h1 l h3
  define at h2
  have h5 : prime a := h2 a h4.left
  have h6 : p = 1 ∨ p = a := dvd_prime h5 h4.right
  disj_syll h6 (prime_not_one h1)
  rewrite [h6]
  show a ∈ l from h4.left
  done

```

The uniqueness of prime factorizations follows from Theorem 7.2.5 of *HTPI*, which says that if two nondecreasing lists of prime numbers have the same product, then the two lists must be the same. In *HTPI*, a key step in the proof of Theorem 7.2.5 is to show that if two nondecreasing lists of prime numbers have the same product, then the last entry of one list is less than or equal to the last entry of the other. In Lean, because of the way the `cons` operation works, it is easier to work with the first entries of the lists.

```

lemma first_le_first {p q : Nat} {l m : List Nat}
  (h1 : nondec_prime_list (p :: l)) (h2 : nondec_prime_list (q :: m))
  (h3 : prod (p :: l) = prod (q :: m)) : p ≤ q := by
  define at h1; define at h2
  have h4 : q ∣ prod (p :: l) := by
    define
    apply Exists.intro (prod m)
    rewrite [←prod_cons]
    show prod (p :: l) = prod (q :: m) from h3
    done
  have h5 : all_prime (q :: m) := h2.left
  rewrite [all_prime_cons] at h5
  have h6 : q ∈ p :: l := prime_in_list h5.left h1.left h4
  have h7 : nondec (p :: l) := h1.right
  rewrite [nondec_cons] at h7
  rewrite [List.mem_cons] at h6
  by_cases on h6
  · -- Case 1. h6 : q = p
    linarith
    done
  · -- Case 2. h6 : q ∈ l
    have h8 : ∀ m ∈ l, p ≤ m := h7.left
    show p ≤ q from h8 q h6
    done
  done

```

The proof of Theorem 7.2.5 is another proof by induction on lists. It uses a few more lemmas whose proofs we leave as exercises.

```

lemma nondec_prime_list_tail {p : Nat} {l : List Nat}
  (h : nondec_prime_list (p :: l)) : nondec_prime_list l := sorry

lemma cons_prod_not_one {p : Nat} {l : List Nat}
  (h : nondec_prime_list (p :: l)) : prod (p :: l) ≠ 1 := sorry

lemma list_nil_iff_prod_one {l : List Nat} (h : nondec_prime_list l) :
  l = [] ↔ prod l = 1 := sorry

lemma prime_pos {p : Nat} (h : prime p) : p > 0 := sorry

theorem Theorem_7_2_5 : ∀ (l1 l2 : List Nat),
  nondec_prime_list l1 → nondec_prime_list l2 →

```

```

prod l1 = prod l2 → l1 = l2 := by
apply List.rec
· -- Base Case. Goal : ∀ (l2 : List Nat), nondec_prime_list [] →
  -- nondec_prime_list l2 → prod [] = prod l2 → [] = l2
  fix l2 : List Nat
  assume h1 : nondec_prime_list []
  assume h2 : nondec_prime_list l2
  assume h3 : prod [] = prod l2
  rewrite [prod_nil, eq_comm, ←list_nil_iff_prod_one h2] at h3
  show [] = l2 from h3.symm
  done
· -- Induction Step
  fix p : Nat
  fix L1 : List Nat
  assume ih : ∀ (L2 : List Nat), nondec_prime_list L1 →
    nondec_prime_list L2 → prod L1 = prod L2 → L1 = L2
  -- Goal : ∀ (l2 : List Nat), nondec_prime_list (p :: L1) →
  -- nondec_prime_list l2 → prod (p :: L1) = prod l2 → p :: L1 = l2
  fix l2 : List Nat
  assume h1 : nondec_prime_list (p :: L1)
  assume h2 : nondec_prime_list l2
  assume h3 : prod (p :: L1) = prod l2
  have h4 : ¬prod (p :: L1) = 1 := cons_prod_not_one h1
  rewrite [h3, ←list_nil_iff_prod_one h2] at h4
  obtain (q : Nat) (h5 : ∃ (L : List Nat), l2 = q :: L) from
    List.exists_cons_of_ne_nil h4
  obtain (L2 : List Nat) (h6 : l2 = q :: L2) from h5
  rewrite [h6] at h2    --h2 : nondec_prime_list (q :: L2)
  rewrite [h6] at h3    --h3 : prod (p :: L1) = prod (q :: L2)
  have h7 : p ≤ q := first_le_first h1 h2 h3
  have h8 : q ≤ p := first_le_first h2 h1 h3.symm
  have h9 : p = q := by linarith
  rewrite [h9, prod_cons, prod_cons] at h3
    --h3 : q * prod L1 = q * prod L2
  have h10 : nondec_prime_list L1 := nondec_prime_list_tail h1
  have h11 : nondec_prime_list L2 := nondec_prime_list_tail h2
  define at h2
  have h12 : all_prime (q :: L2) := h2.left
  rewrite [all_prime_cons] at h12
  have h13 : q > 0 := prime_pos h12.left
  have h14 : prod L1 = prod L2 := Nat.eq_of_mul_eq_mul_left h13 h3
  have h15 : L1 = L2 := ih L2 h10 h11 h14

```

```

rewrite [h6, h9, h15]
rfl
done
done

```

Putting it all together, we can finally prove the fundamental theorem of arithmetic, which is stated as Theorem 7.2.6 in *HTPI*:

```

theorem fund_thm_arith (n : Nat) (h : n ≥ 1) :
  ∃! (l : List Nat), prime_factorization n l := by
  exists_unique
  · -- Existence
    show ∃ (l : List Nat), prime_factorization n l from
      exists_prime_factorization n h
    done
  · -- Uniqueness
    fix l1 : List Nat; fix l2 : List Nat
    assume h1 : prime_factorization n l1
    assume h2 : prime_factorization n l2
    define at h1; define at h2
    have h3 : prod l1 = n := h1.right
    rewrite [←h2.right] at h3
    show l1 = l2 from Theorem_7_2_5 l1 l2 h1.left h2.left h3
    done
done

```

Exercises

1.

```
lemma dvd_prime {a p : Nat}
  (h1 : prime p) (h2 : a | p) : a = 1 ∨ a = p := sorry
```
2.

```
--Hints: Start with apply List.rec.
--You may find the theorem mul_ne_zero useful.
theorem prod_nonzero_nonzero : ∀ (l : List Nat),
  (∀ a ∈ l, a ≠ 0) → prod l ≠ 0 := sorry
```
3.

```
theorem rel_prime_iff_no_common_factor (a b : Nat) :
  rel_prime a b ↔ ¬∃ (p : Nat), prime p ∧ p | a ∧ p | b := sorry
```
4.

```
theorem rel_prime_symm {a b : Nat} (h : rel_prime a b) :
  rel_prime b a := sorry
```

5.

```
lemma in_prime_factorization_iff_prime_factor {a : Nat} {l : List Nat}
    (h1 : prime_factorization a l) (p : Nat) :
    p ∈ l ↔ prime_factor p a := sorry
```
6.

```
theorem Exercise_7_2_5 {a b : Nat} {l m : List Nat}
    (h1 : prime_factorization a l) (h2 : prime_factorization b m) :
    rel_prime a b ↔ (∃ (p : Nat), p ∈ l ∧ p ∈ m) := sorry
```
7.

```
theorem Exercise_7_2_6 (a b : Nat) :
    rel_prime a b ↔ ∃ (s t : Int), s * a + t * b = 1 := sorry
```
8.

```
theorem Exercise_7_2_7 {a b a' b' : Nat}
    (h1 : rel_prime a b) (h2 : a' ∣ a) (h3 : b' ∣ b) :
    rel_prime a' b' := sorry
```
9.

```
theorem Exercise_7_2_9 {a b j k : Nat}
    (h1 : gcd a b ≠ 0) (h2 : a = j * gcd a b) (h3 : b = k * gcd a b) :
    rel_prime j k := sorry
```
10.

```
theorem Exercise_7_2_17a (a b c : Nat) :
    gcd a (b * c) ∣ gcd a b * gcd a c := sorry
```

7.3. Modular Arithmetic

If m is a positive integer and a and b are integers, then *HTPI* uses the notation $a \equiv b \pmod{m}$, or sometimes $a \equiv_m b$, to indicate that a is congruent to b modulo m , which is defined to mean $m \mid (a - b)$. Congruence modulo m is an equivalence relation on the integers, and therefore it induces a partition \mathbb{Z}/\equiv_m of the integers, as shown in Section 4.5 of *HTPI*. The elements of this partition are the equivalence classes $[a]_m$ for $a \in \mathbb{Z}$; we will call these *congruence classes modulo m* . Section 7.3 of *HTPI* defines operations of addition and multiplication of congruence classes and proves algebraic properties of those operations.

For the purpose of working out the rules of modular arithmetic, the only important properties of congruence classes are the following:

1. For every integer a , there is a corresponding congruence class $[a]_m \in \mathbb{Z}/\equiv_m$.
2. For every congruence class $X \in \mathbb{Z}/\equiv_m$, there is some integer a such that $X = [a]_m$.
3. For all integers a and b , $[a]_m = [b]_m$ if and only if $a \equiv_m b$.
4. For all integers a and b , $[a]_m + [b]_m = [a + b]_m$.
5. For all integers a and b , $[a]_m \cdot [b]_m = [ab]_m$.

To study congruence modulo m in Lean, we will declare m to have type `Nat`, which allows for the possibility that $m = 0$, but we will mostly focus on the case $m \neq 0$. For a and b of type `Int`, we define `congr_mod m a b` to mean that a is congruent to b modulo m . Notice that to define this relation in Lean, we must coerce m to be an integer so that we can use it in the divisibility relation on the integers.

```
def congr_mod (m : Nat) (a b : Int) : Prop := ( $\uparrow m$  : Int) | (a - b)
```

We can teach Lean to use more familiar notation for congruence modulo m by giving the following command:

```
notation:50 a " ≡ " b " (MOD " m ")" => congr_mod m a b
```

This tells Lean that if we type $a \equiv b \pmod{m}$, then Lean should interpret it as `congr_mod m a b`. (To enter the symbol \equiv , type `\equiv`. Don't worry about the `:50` in the `notation` command above. It is there to help Lean parse this new notation when it occurs together with other notation.)

We can now prove that congruence modulo m is reflexive, symmetric, and transitive. In these proofs, we leave it to Lean to fill in coercions when they are necessary, and as usual, we leave some details as exercises.

```
theorem congr_refl (m : Nat) : ∀ (a : Int), a ≡ a (MOD m) := sorry

theorem congr_symm {m : Nat} : ∀ {a b : Int},
  a ≡ b (MOD m) → b ≡ a (MOD m) := by
  fix a : Int; fix b : Int
  assume h1 : a ≡ b (MOD m)
  define at h1           --h1 : ∃ (c : Int), a - b =  $\uparrow m * c$ 
  define                 --Goal : ∃ (c : Int), b - a =  $\uparrow m * c$ 
  obtain (c : Int) (h2 : a - b = m * c) from h1
  apply Exists.intro (-c)
  show b - a = m * (-c) from
    calc b - a
      _ = -(a - b) := by ring
      _ = -(m * c) := by rw [h2]
      _ = m * (-c) := by ring
  done

theorem congr_trans {m : Nat} : ∀ {a b c : Int},
  a ≡ b (MOD m) → b ≡ c (MOD m) → a ≡ c (MOD m) := sorry
```

We could now repeat the entire development of \mathbb{Z}/\equiv_m in Lean, but there is no need to do so; such a development is already included in Lean's library of definitions and theorems. For each natural number m , Lean has a type $\mathbf{ZMod}\ m$, and the objects of that type are Lean's version of the congruence classes modulo m . We should warn you that Lean's way of defining $\mathbf{ZMod}\ m$ differs in some ways from *HTPI*'s definition of \mathbb{Z}/\equiv_m . In particular, objects of type $\mathbf{ZMod}\ m$ are not sets of integers. Thus, if x has type $\mathbf{ZMod}\ m$ and a has type \mathbf{Int} , then Lean will not understand what you mean if you write $a \in x$. However, Lean's congruence classes have the properties 1–5 listed above, and that's all that will matter to us.

Property 1 says that if a has type \mathbf{Int} , then there should be a corresponding congruence class in $\mathbf{ZMod}\ m$. In fact, you can find the corresponding congruence class by simply coercing a to have type $\mathbf{ZMod}\ m$. But for the sake of clarity, we will introduce a function for computing the congruence class modulo m of a :

```
def cc (m : Nat) (a : Int) : ZMod m := (↑a : ZMod m)
```

Thus, $cc\ m\ a$ is the congruence class modulo m of a . Once again, it will be convenient to teach Lean to use more familiar notation for congruence classes, so we give the command:

```
notation:max "[" a "]_m" m:max => cc m a
```

Now if we type $[a]_m$, then Lean will interpret it as $cc\ m\ a$. Thus, from now on, $[a]_m$ will be our notation in Lean for the congruence class modulo m of a ; it corresponds to the *HTPI* notation $[a]_m$. (Once again, you can ignore the two occurrences of $:max$ in the `notation` command above.)

Properties 2–5 of congruence classes are established by the following theorems:

```
theorem cc_rep {m : Nat} (x : ZMod m) : ∃ (a : Int), x = [a]_m

theorem cc_eq_iff_congr (m : Nat) (a b : Int) :
  [a]_m = [b]_m ↔ a ≡ b (MOD m)

theorem add_class (m : Nat) (a b : Int) :
  [a]_m + [b]_m = [a + b]_m

theorem mul_class (m : Nat) (a b : Int) :
  [a]_m * [b]_m = [a * b]_m
```

We won't discuss the proofs of these theorems, since they depend on details of Lean's representation of objects of type $\mathbf{ZMod}\ m$ that are beyond the scope of this book. But these theorems are all we will need to use to develop the theory of modular arithmetic.

7.3. Modular Arithmetic

In many of our theorems about `ZMod m`, we will need to include a hypothesis that $m \neq 0$. We will usually state this hypothesis in the form `NeZero m`, which is a proposition that is equivalent to $m \neq 0$. Indeed, there is a theorem in Lean's library that asserts this equivalence:

```
@neZero_iff : ∀ {R : Type u_1} [inst : Zero R] {n : R},
  NeZero n ↔ n ≠ 0
```

What distinguishes `NeZero m` from $m \neq 0$ is that `NeZero m` is what is called a *type class*. What this means is that, once Lean has a proof of `NeZero m` for some natural number m , it will remember that proof and be able to recall it when necessary. As a result, `NeZero m` can be used as a new kind of implicit argument in the statement of a theorem. To make it an implicit argument, we write it in square brackets, like this: `[NeZero m]`. When applying a theorem that includes this hypothesis, there is no need to supply a proof that $m \neq 0$; as long as Lean knows about such a proof, it will recall that proof on its own. When you are proving a theorem that includes the hypothesis `[NeZero m]`, Lean will recognize `NeZero.ne m` as a proof that $m \neq 0$.

The first theorem in Section 7.3 of *HTPI*, Theorem 7.3.1, says that if $m \neq 0$, then every integer a is congruent modulo m to exactly one integer r satisfying $0 \leq r < m$. As explained in *HTPI*, we say that $\{0, 1, \dots, m - 1\}$ is a *complete residue system modulo m*. This implies that the objects of type `ZMod m` are precisely the congruence classes $[0]_m, [1]_m, \dots, [m - 1]_m$.

The proof of Theorem 7.3.1 makes use of the quotient and remainder when a is divided by m . In Section 6.4, we learned about the Lean theorems `Nat.div_add_mod` and `Nat.mod_lt`, but those theorems concerned quotients and remainders when dividing *natural numbers*. Fortunately, Lean has similar theorems for dealing with division of integers:

```
Int.ediv_add_emod : ∀ (a b : ℤ), b * (a / b) + a % b = a
Int.emod_lt_of_pos : ∀ (a : ℤ) {b : ℤ}, 0 < b → a % b < b
Int.emod_nonneg : ∀ (a : ℤ) {b : ℤ}, b ≠ 0 → 0 ≤ a % b
```

We now have all the background we need to prove Theorem 7.3.1 in Lean. We begin with a few lemmas, some of which require the hypothesis `[NeZero m]`. The proof of the theorem closely follows the proof in *HTPI*. Note that all of the proofs below involve the expression $a \% m$. Since a is an integer, the operator `%` in this expression must be the integer version of the mod operator, and therefore m must be coerced to be an integer. Thus, Lean interprets $a \% m$ as $a \% \uparrow m$

```
lemma mod_nonneg (m : Nat) [NeZero m] (a : Int) : 0 ≤ a % m := by
  have h1 : (\uparrow m : Int) ≠ 0 := (Nat.cast_ne_zero).rtl (NeZero.ne m)
  show 0 ≤ a % m from Int.emod_nonneg a h1
  done

lemma mod_lt (m : Nat) [NeZero m] (a : Int) : a % m < m := sorry
```

```

lemma congr_mod_mod (m : Nat) (a : Int) : a ≡ a % m (MOD m) := by
  define
  have h1 : m * (a / m) + a % m = a := Int.ediv_add_emod a m
  apply Exists.intro (a / m)
  show a - a % m = m * (a / m) from
    calc a - (a % m)
      _ = m * (a / m) + a % m - a % m := by rw [h1]
      _ = m * (a / m) := by ring
  done

lemma mod_cmpl_res (m : Nat) [NeZero m] (a : Int) :
  0 ≤ a % m ∧ a % m < m ∧ a ≡ a % m (MOD m) :=
  And.intro (mod_nonneg m a) (And.intro (mod_lt m a) (congr_mod_mod m a))

theorem Theorem_7_3_1 (m : Nat) [NeZero m] (a : Int) :
  ∃! (r : Int), 0 ≤ r ∧ r < m ∧ a ≡ r (MOD m) := by
  exists_unique
  · -- Existence
    apply Exists.intro (a % m)
    show 0 ≤ a % m ∧ a % m < m ∧ a ≡ a % m (MOD m)
      from mod_cmpl_res m a
    done
  · -- Uniqueness
    fix r1 : Int; fix r2 : Int
    assume h1 : 0 ≤ r1 ∧ r1 < m ∧ a ≡ r1 (MOD m)
    assume h2 : 0 ≤ r2 ∧ r2 < m ∧ a ≡ r2 (MOD m)
    have h3 : r1 ≡ r2 (MOD m) :=
      congr_trans (congr_symm h1.right.right) h2.right.right
    obtain (d : Int) (h4 : r1 - r2 = m * d) from h3
    have h5 : r1 - r2 < m * 1 := by linarith
    have h6 : m * (-1) < r1 - r2 := by linarith
    rewrite [h4] at h5 --h5 : m * d < m * 1
    rewrite [h4] at h6 --h6 : m * -1 < m * d
    have h7 : (↑m : Int) ≥ 0 := Nat.cast_nonneg m
    have h8 : d < 1 := lt_of_mul_lt_mul_of_nonneg_left h5 h7
    have h9 : -1 < d := lt_of_mul_lt_mul_of_nonneg_left h6 h8
    have h10 : d = 0 := by linarith
    show r1 = r2 from
      calc r1
        _ = r1 - r2 + r2 := by ring
        _ = m * 0 + r2 := by rw [h4, h10]
        _ = r2 := by ring

```

```
done
done
```

The lemma `mod_cmpl_res` above says that $a \% m$ is an element of the complete residue system $\{0, 1, \dots, m - 1\}$ that is congruent to a modulo m . The lemma requires the hypothesis `[NeZero m]`, because its proof appeals to two previous lemmas, `mod_nonneg` and `mod_lt`, that require that hypothesis. But when the proof invokes those previous lemmas, this hypothesis is not mentioned, because it is an implicit argument.

An immediate consequence of the lemma `congr_mod_mod` is the following lemma, which will be useful to us later.

```
lemma cc_eq_mod (m : Nat) (a : Int) : [a]_m = [a \% m]_m :=
  (cc_eq_iff_congr m a (a \% m)).rtl (congr_mod_mod m a)
```

Theorem 7.3.6 in *HTPI* states a number of algebraic properties of modular arithmetic. These properties all follow easily from the theorems we have already stated. To illustrate this, we prove two parts of the theorem.

```
theorem Theorem_7_3_6_1 {m : Nat} (X Y : ZMod m) : X + Y = Y + X := by
  obtain (a : Int) (h1 : X = [a]_m) from cc_rep X
  obtain (b : Int) (h2 : Y = [b]_m) from cc_rep Y
  rewrite [h1, h2]
  have h3 : a + b = b + a := by ring
  show [a]_m + [b]_m = [b]_m + [a]_m from
    calc [a]_m + [b]_m
      _ = [a + b]_m := add_class m a b
      _ = [b + a]_m := by rw [h3]
      _ = [b]_m + [a]_m := (add_class m b a).symm
  done

theorem Theorem_7_3_6_7 {m : Nat} (X : ZMod m) : X * [1]_m = X := by
  obtain (a : Int) (h1 : X = [a]_m) from cc_rep X
  rewrite [h1]
  have h2 : a * 1 = a := by ring
  show [a]_m * [1]_m = [a]_m from
    calc [a]_m * [1]_m
      _ = [a * 1]_m := mul_class m a 1
      _ = [a]_m := by rw [h2]
  done
```

Theorem 7.3.6.7 shows that $[1]_m$ is the multiplicative identity element for $\mathbb{Z}\text{Mod } m$. We say that a congruence class Y is a *multiplicative inverse* of another class X if $X * Y = [1]_m$, and a congruence class is *invertible* if it has a multiplicative inverse:

```
def invertible {m : Nat} (X : ZMod m) : Prop :=
  ∃ (Y : ZMod m), X * Y = [1]_m
```

Which congruence classes are invertible? The answer is given by Theorem 7.3.7 in *HTPI*, which says that if a is a positive integer, then $[a]_m$ is invertible if and only if m and a are relatively prime. The proof uses an exercise from the last section. Here is the Lean version of the proof.

```
theorem Exercise_7_2_6 (a b : Nat) :
  rel_prime a b ↔ ∃ (s t : Int), s * a + t * b = 1 := sorry

lemma gcd_c2_inv {m a : Nat} (h1 : rel_prime m a) :
  [a]_m * [gcd_c2 m a]_m = [1]_m := by
  set s : Int := gcd_c1 m a
  have h2 : s * m + (gcd_c2 m a) * a = gcd m a := gcd_lin_comb a m
  define at h1
  rewrite [h1, Nat.cast_one] at h2 --h2 : s * ↑m + gcd_c2 m a * ↑a = 1
  rewrite [mul_class, cc_eq_iff_congr]
  define --Goal : ∃ (c : Int), ↑a * gcd_c2 m a - 1 = ↑m * c
  apply Exists.intro (-s)
  show a * (gcd_c2 m a) - 1 = m * (-s) from
  calc a * (gcd_c2 m a) - 1
    = s * m + (gcd_c2 m a) * a + m * (-s) - 1 := by ring
    = 1 + m * (-s) - 1 := by rw [h2]
    = m * (-s) := by ring
  done

theorem Theorem_7_3_7 (m a : Nat) :
  invertible [a]_m ↔ rel_prime m a := by
  apply Iff.intro
  · -- (→)
    assume h1 : invertible [a]_m
    define at h1
    obtain (Y : ZMod m) (h2 : [a]_m * Y = [1]_m) from h1
    obtain (b : Int) (h3 : Y = [b]_m) from cc_rep Y
    rewrite [h3, mul_class, cc_eq_iff_congr] at h2
    define at h2
    obtain (c : Int) (h4 : a * b - 1 = m * c) from h2
```

```

rewrite [Exercise_7_2_6]
  --Goal : ∃ (s t : Int), s * m + t * a = 1
apply Exists.intro (-c)
apply Exists.intro b
show (-c) * m + b * a = 1 from
  calc (-c) * m + b * a
    _ = (-c) * m + (a * b - 1) + 1 := by ring
    _ = (-c) * m + m * c + 1 := by rw [h4]
    _ = 1 := by ring
done
· -- (→)
assume h1 : rel_prime m a
define
show ∃ (Y : ZMod m), [a]_m * Y = [1]_m from
  Exists.intro [gcd_c2 m a]_m (gcd_c2_inv h1)
done
done

```

Exercises

1. `theorem congr_trans {m : Nat} : ∀ {a b c : Int},
 a ≡ b (MOD m) → b ≡ c (MOD m) → a ≡ c (MOD m) := sorry`
2. `theorem Theorem_7_3_6_3 {m : Nat} (X : ZMod m) : X + [0]_m = X := sorry`
3. `theorem Theorem_7_3_6_4 {m : Nat} (X : ZMod m) :
 ∃ (Y : ZMod m), X + Y = [0]_m := sorry`
4. `theorem Exercise_7_3_4a {m : Nat} (Z1 Z2 : ZMod m)
 (h1 : ∀ (X : ZMod m), X + Z1 = X)
 (h2 : ∀ (X : ZMod m), X + Z2 = X) : Z1 = Z2 := sorry`
5. `theorem Exercise_7_3_4b {m : Nat} (X Y1 Y2 : ZMod m)
 (h1 : X + Y1 = [0]_m) (h2 : X + Y2 = [0]_m) : Y1 = Y2 := sorry`
6. `theorem Theorem_7_3_10 (m a : Nat) (b : Int) :
 ¬(↑(gcd m a) : Int) | b → ¬∃ (x : Int), a * x ≡ b (MOD m) := sorry`
7. `theorem Theorem_7_3_11 (m n : Nat) (a b : Int) (h1 : n ≠ 0) :
 n * a ≡ n * b (MOD n * m) ↔ a ≡ b (MOD m) := sorry`
8. `theorem Exercise_7_3_16 {m : Nat} {a b : Int} (h : a ≡ b (MOD m)) :
 ∀ (n : Nat), a ^ n ≡ b ^ n (MOD m) := sorry`

```

9. example {m : Nat} [NeZero m] (X : ZMod m) :
  ∃! (a : Int), 0 ≤ a ∧ a < m ∧ X = [a]_m := sorry

10. theorem congr_rel_prime {m a b : Nat} (h1 : a ≡ b (MOD m)) :
  rel_prime m a ↔ rel_prime m b := sorry

11. --Hint: You may find the theorem Int.ofNat_mod_ofNat useful.
theorem rel_prime_mod (m a : Nat) :
  rel_prime m (a % m) ↔ rel_prime m a := sorry

12. lemma congr_iff_mod_eq_Int (m : Nat) (a b : Int) [NeZero m] :
  a ≡ b (MOD m) ↔ a % ↑m = b % ↑m := sorry

--Hint for next theorem: Use the lemma above,
--together with the theorems Int.ofNat_mod_ofNat and Nat.cast_inj.
theorem congr_iff_mod_eq_Nat (m a b : Nat) [NeZero m] :
  ↑a ≡ ↑b (MOD m) ↔ a % m = b % m := sorry

```

7.4. Euler's Theorem

The main result of Section 7.4 of *HTPI* is Euler's theorem. The statement of the theorem involves Euler's totient function φ . For any positive integer m , *HTPI* defines $\varphi(m)$ to be the number of elements of \mathbb{Z}/\equiv_m that have multiplicative inverses. In order to state and prove Euler's theorem in Lean, our first task is to define a Lean function `phi : Nat → Nat` that computes the totient function.

Since $\{0, 1, \dots, m - 1\}$ is a complete residue system modulo m , `phi m` can be described as the number of natural numbers $a < m$ such that $[a]_m$ is invertible. According to [Theorem_7_3_7](#), $[a]_m$ is invertible if and only if m and a are relatively prime, so `phi m` is also equal to the number of natural numbers $a < m$ that are relatively prime to m . We begin by defining a function `num_rp_below m k` that counts the number of natural numbers less than k that are relatively prime to m .

```

def num_rp_below (m k : Nat) : Nat :=
  match k with
  | 0 => 0
  | j + 1 => if gcd m j = 1 then (num_rp_below m j) + 1
               else num_rp_below m j

```

This is the first time we have used an `if ... then ... else` expression in a Lean definition. To prove theorems about such expressions, we will need two theorems from Lean's library, `if_pos` and `if_neg`. The `#check` command tells us what they say:

7.4. Euler's Theorem

```
@if_pos : ∀ {c : Prop} {h : Decidable c},
  c → ∀ {α : Sort u_1} {t e : α}, (if c then t else e) = t

@if_neg : ∀ {c : Prop} {h : Decidable c},
  ¬c → ∀ {α : Sort u_1} {t e : α}, (if c then t else e) = e
```

Ignoring the implicit arguments, this tells us that if hc is a proof of a proposition c , then $\text{if_pos } hc$ is a proof of $(\text{if } c \text{ then } t \text{ else } e) = t$, and if $h\neg c$ is a proof of $\neg c$, then $\text{if_neg } h\neg c$ is a proof of $(\text{if } c \text{ then } t \text{ else } e) = e$. (Technically, the implicit arguments say that c must be a “decidable” proposition, but we won’t worry about that detail.) We often use these theorems to evaluate an expression of the form $\text{if } c \text{ then } t \text{ else } e$ as either t or e , depending on whether c is true or false.

```
lemma num_rp_below_base {m : Nat} :
  num_rp_below m 0 = 0 := by rfl

lemma num_rp_below_step_rp {m j : Nat} (h : rel_prime m j) :
  num_rp_below m (j + 1) = (num_rp_below m j) + 1 := if_pos h

lemma num_rp_below_step_not_rp {m j : Nat} (h : ¬rel_prime m j) :
  num_rp_below m (j + 1) = num_rp_below m j := if_neg h
```

We can now use `num_rp_below` to define the totient function.

```
def phi (m : Nat) : Nat := num_rp_below m m

lemma phi_def (m : Nat) : phi m = num_rp_below m m := by rfl

#eval phi 10 --Answer: 4
```

With this preparation, we can now state the theorem we will prove:

```
theorem Theorem_7_4_2 {m a : Nat} [NeZero m] (h1 : rel_prime m a) :
  [a]_m ^ (phi m) = [1]_m
```

In preparation for proving this theorem, *HTPI* first shows that the set of invertible congruence classes is closed under inverses and multiplication. For our purposes, we will find it useful to prove a slightly different lemma. Note that Lean knows all of the basic algebraic laws of addition and multiplication of congruence classes, and as a result the `ring` tactic can be used to do algebraic reasoning in `ZMod m`, as illustrated in the proof below.

```

lemma prod_inv_iff_inv {m : Nat} {X : ZMod m}
  (h1 : invertible X) (Y : ZMod m) :
  invertible (X * Y) ↔ invertible Y := by
  apply Iff.intro
  · -- (→)
    assume h2 : invertible (X * Y)
    obtain (Z : ZMod m) (h3 : X * Y * Z = [1]_m) from h2
    apply Exists.intro (X * Z)
    rewrite [←h3] --Goal : Y * (X * Z) = X * Y * Z
    ring      --Note that ring can do algebra in ZMod m
    done
  · -- (←)
    assume h2 : invertible Y
    obtain (Xi : ZMod m) (h3 : X * Xi = [1]_m) from h1
    obtain (Yi : ZMod m) (h4 : Y * Yi = [1]_m) from h2
    apply Exists.intro (Xi * Yi)
    show (X * Y) * (Xi * Yi) = [1]_m from
      calc X * Y * (Xi * Yi)
        _ = (X * Xi) * (Y * Yi) := by ring
        _ = [1]_m * [1]_m := by rw [h3, h4]
        _ = [1]_m := Theorem_7_3_6_7 [1]_m
    done
  done

```

One of the key ideas in the proof of Theorem 7.4.2 in *HTPI* involves computing the product of all invertible congruence classes. To compute this product in Lean, we begin by defining a function $F : \mathbb{N} \rightarrow \text{ZMod } m$ as follows:

```

def F (m i : Nat) : ZMod m := if gcd m i = 1 then [i]_m else [1]_m

lemma F_rp_def {m i : Nat} (h : rel_prime m i) :
  F m i = [i]_m := if_pos h

lemma F_not_rp_def {m i : Nat} (h : ¬rel_prime m i) :
  F m i = [1]_m := if_neg h

```

Note that F is defined as a function of two natural numbers, m and i , but as we discussed in Section 5.4, it follows that the partial application $F m$ is a function from \mathbb{N} to $\text{ZMod } m$.

Now consider the product $(F m 0) * (F m 1) * \dots * (F m (m - 1))$; we will call this the *F-product*. If m and i are not relatively prime, then $F m i = [1]_m$, and since $[1]_m$ is the multiplicative identity element of $\text{ZMod } m$, the factor $F m i$ contributes nothing to the product.

7.4. Euler's Theorem

Thus, the F -product is equal to the product of the factors $F_m i$ for which m and i are relatively prime. But for those values of i , $F_m i = [i]_m$, so the product is equal to the product of all congruence classes $[i]_m$ with m and i relatively prime. By Theorem 7.3.7, that is the product of all invertible congruence classes.

To express the F -product in Lean, we imitate our approach to summations, as described in Chapter 6. We begin by defining `prod_seq j k f` to be the product of a sequence of j consecutive values of the function f , starting with $f k$:

```
def prod_seq {m : Nat}
  (j k : Nat) (f : Nat → ZMod m) : ZMod m :=
match j with
| 0 => [1]_m
| n + 1 => prod_seq n k f * f (k + n)

lemma prod_seq_base {m : Nat}
  (k : Nat) (f : Nat → ZMod m) : prod_seq 0 k f = [1]_m := by rfl

lemma prod_seq_step {m : Nat}
  (n k : Nat) (f : Nat → ZMod m) :
  prod_seq (n + 1) k f = prod_seq n k f * f (k + n) := by rfl

lemma prod_seq_zero_step {m : Nat}
  (n : Nat) (f : Nat → ZMod m) :
  prod_seq (n + 1) 0 f = prod_seq n 0 f * f n := sorry
```

Using this notation, the expression `prod_seq m 0 (F m)` denotes the F -product, which, as we saw earlier, is equal to the product of the invertible congruence classes.

Now suppose a is a natural number that is relatively prime to m . The next step in the proof in *HTPI* is to multiply each factor in the product of the invertible congruence classes by $[a]_m$. To do this, we define a function G as follows:

```
def G (m a i : Nat) : Nat := (a * i) % m
```

Consider the following product, which we will call the *FG-product*:

$$(F_m (G_m a_0)) * (F_m (G_m a_1)) * \dots * (F_m (G_m a_{(m-1)})).$$

Using the partial application $G_m a$, which is a function from `Nat` to `Nat`, we can express this in Lean as `prod_seq m 0 ((F m) ∘ (G m a))`. To understand this product we will need two facts about G .

```

lemma cc_G (m a i : Nat) : [G m a i]_m = [a]_m * [i]_m :=
calc [G m a i]_m =
_ = [(a * i) % m]_m := by rfl
_ = [a * i]_m := (cc_eq_mod m (a * i)).symm
_ = [a]_m * [i]_m := (mul_class m a i).symm

lemma G_rp_iff {m a : Nat} (h1 : rel_prime m a) (i : Nat) :
rel_prime m (G m a i) ↔ rel_prime m i := by
have h2 : invertible [a]_m := (Theorem_7_3_7 m a).rtl h1
show rel_prime m (G m a i) ↔ rel_prime m i from
calc rel_prime m (G m a i)
_ ↔ invertible [G m a i]_m := (Theorem_7_3_7 m (G m a i)).symm
_ ↔ invertible ([a]_m * [i]_m) := by rw [cc_G]
_ ↔ invertible [i]_m := prod_inv_iff_inv h2 ([i]_m)
_ ↔ rel_prime m i := Theorem_7_3_7 m i
done

```

Now let's analyze the FG-product. If i is not relatively prime to m , then by G_{rp_iff} , $G m a i$ is also not relatively prime to m , so $F m (G m a i) = [1]_m$. As before, this means that these terms contribute nothing to the FG-product. If i is relatively prime to m , then so is $G m a i$, and therefore, by cc_G ,

$$F m (G m a i) = [G m a i]_m = [a]_m * [i]_m = [a]_m * (F m i).$$

This means that, in the FG-product, each factor contributed by a value of i that is relatively prime to m is $[a]_m$ times the corresponding factor in the F-product. Since the number of such factors is $\phi(m)$, it follows that the FG-product is $[a]_m^{\phi(m)}$ times the F-product.

Let's see if we can prove this in Lean.

```

lemma FG_rp {m a i : Nat} (h1 : rel_prime m a) (h2 : rel_prime m i) :
F m (G m a i) = [a]_m * F m i := by
have h3 : rel_prime m (G m a i) := (G_rp_iff h1 i).rtl h2
show F m (G m a i) = [a]_m * F m i from
calc F m (G m a i)
_ = [G m a i]_m := F_rp_def h3
_ = [a]_m * [i]_m := cc_G m a i
_ = [a]_m * F m i := by rw [F_rp_def h2]
done

lemma FG_not_rp {m a i : Nat} (h1 : rel_prime m a) (h2 : ¬rel_prime m i) :
F m (G m a i) = [1]_m := sorry

```

```


lemma FG_prod {m a : Nat} (h1 : rel_prime m a) :
  ∀ (k : Nat), prod_seq k 0 ((F m) ∘ (G m a)) =
    [a]_m ^ (num_rp_below m k) * prod_seq k 0 (F m) := by
  by_induc
  · -- Base Case
    show prod_seq 0 0 ((F m) ∘ (G m a)) =
      [a]_m ^ (num_rp_below m 0) * prod_seq 0 0 (F m) from
    calc prod_seq 0 0 ((F m) ∘ (G m a))
      _ = [1]_m := prod_seq_base _ _
      _ = [a]_m ^ 0 * [1]_m := by ring
      _ = [a]_m ^ (num_rp_below m 0) * prod_seq 0 0 (F m) := by
        rw [num_rp_below_base, prod_seq_base]
    done
  · -- Induction Step
    fix k : Nat
    assume ih : prod_seq k 0 ((F m) ∘ (G m a)) =
      [a]_m ^ (num_rp_below m k) * prod_seq k 0 (F m)
    by_cases h2 : rel_prime m k
    · -- Case 1. h2 : rel_prime m k
      show prod_seq (k + 1) 0 ((F m) ∘ (G m a)) =
        [a]_m ^ (num_rp_below m (k + 1)) *
        prod_seq (k + 1) 0 (F m) from
      calc prod_seq (k + 1) 0 ((F m) ∘ (G m a))
        _ = prod_seq k 0 ((F m) ∘ (G m a)) *
          F m (G m a k) := prod_seq_zero_step _ _
        _ = [a]_m ^ (num_rp_below m k) * prod_seq k 0 (F m) *
          F m (G m a k) := by rw [ih]
        _ = [a]_m ^ (num_rp_below m k) * prod_seq k 0 (F m) *
          ([a]_m * F m k) := by rw [FG_rp h1 h2]
        _ = [a]_m ^ ((num_rp_below m k) + 1) *
          ((prod_seq k 0 (F m)) * F m k) := by ring
        _ = [a]_m ^ (num_rp_below m (k + 1)) *
          prod_seq (k + 1) 0 (F m) := by
            rw [num_rp_below_step_rp h2, prod_seq_zero_step]
      done
    · -- Case 2. h2 : ¬rel_prime m k
      show prod_seq (k + 1) 0 ((F m) ∘ (G m a)) =
        [a]_m ^ (num_rp_below m (k + 1)) *
        prod_seq (k + 1) 0 (F m) from
      calc prod_seq (k + 1) 0 ((F m) ∘ (G m a))
        _ = prod_seq k 0 ((F m) ∘ (G m a)) *
          F m (G m a k) := prod_seq_zero_step _ _


```

```

_ = [a]_m ^ (num_rp_below m k) * prod_seq k 0 (F m) *
    F m (G m a k) := by rw [ih]
_ = [a]_m ^ (num_rp_below m k) * prod_seq k 0 (F m) *
    ([1]_m) := by rw [FG_not_rp h1 h2]
_ = [a]_m ^ (num_rp_below m k) *
    (prod_seq k 0 (F m) * ([1]_m)) := by ring
_ = [a]_m ^ (num_rp_below m (k + 1)) *
    prod_seq (k + 1) 0 (F m) := by
    rw [num_rp_below_step_not_rp h2, prod_seq_zero_step,
        F_not_rp_def h2]
done
done
done

```

The lemma `FG_prod`, in the case $k = m$, tells us that

$$\text{prod_seq } m \ 0 ((F \ m) \circ (G \ m \ a)) = [a]_m ^ (\phi(m)) * \text{prod_seq } m \ 0 (F \ m).$$

In other words, we have proven that the `FG`-product is $[a]_m ^ (\phi(m))$ times the `F`-product.

And now we come to the central idea in the proof of [Theorem_7_4_2](#): the congruence classes that are multiplied in the `FG`-product are exactly the same as the congruence classes multiplied in the `F`-product, but listed in a different order. The reason for this is that the function `G m a` permutes the natural numbers less than m . Since multiplication of congruence classes is commutative and associative, it follows that the `FG`-product and the `F`-product are equal.

To prove these claims, we first define what it means for a function to permute the natural numbers less than a natural number n .

```

def maps_below (n : Nat) (g : Nat → Nat) : Prop := ∀ i < n, g i < n

def one_one_below (n : Nat) (g : Nat → Nat) : Prop :=
  ∀ i1 < n, ∀ i2 < n, g i1 = g i2 → i1 = i2

def onto_below (n : Nat) (g : Nat → Nat) : Prop :=
  ∀ k < n, ∃ i < n, g i = k

def perm_below (n : Nat) (g : Nat → Nat) : Prop :=
  maps_below n g ∧ one_one_below n g ∧ onto_below n g

```

The proofs of our next two lemmas are somewhat long. We state them now, but put off discussion of their proofs until the end of the section. The first lemma says that, if m and a are relatively prime, then $G m a$ permutes the natural numbers less than m , and the second says that permuting the terms of a product does not change the value of the product.

```

lemma G_perm_below {m a : Nat} [NeZero m]
  (h1 : rel_prime m a) : perm_below m (G m a)

lemma perm_prod {m : Nat} (f : Nat → ZMod m) :
  ∀ (n : Nat), ∀ (g : Nat → Nat), perm_below n g →
    prod_seq n 0 f = prod_seq n 0 (f ∘ g)

```

There is just one more fact we need before we can prove `Theorem_7_4_2`: all of the factors in the F -product are invertible, and therefore the F -product is invertible:

```

lemma F_invertible (m i : Nat) : invertible (F m i) := by
  by_cases h : rel_prime m i
  . -- Case 1. h : rel_prime m i
    rewrite [F_rp_def h]
    show invertible [i]_m from (Theorem_7_3_7 m i).rtl h
    done
  . -- Case 2. h : ¬rel_prime m i
    rewrite [F_not_rp_def h]
    apply Exists.intro [1]_m
    show [1]_m * [1]_m = [1]_m from Theorem_7_3_6_7 [1]_m
    done
  done

lemma Fprod_invertible (m : Nat) :
  ∀ (k : Nat), invertible (prod_seq k 0 (F m)) := by
  by_induc
  . -- Base Case
    apply Exists.intro [1]_m
    show prod_seq 0 0 (F m) * [1]_m = [1]_m from
      calc prod_seq 0 0 (F m) * [1]_m
        _ = [1]_m * [1]_m := by rw [prod_seq_base]
        _ = [1]_m := Theorem_7_3_6_7 ([1]_m)
    done
  . -- Induction Step
    fix k : Nat
    assume ih : invertible (prod_seq k 0 (F m))
    rewrite [prod_seq_zero_step]
    show invertible (prod_seq k 0 (F m) * (F m k)) from
      (prod_inv_iff_inv ih (F m k)).rtl (F_invertible m k)
    done
  done

```

7.4. Euler's Theorem

We now have everything we need to prove `Theorem_7_4_2`:

```
theorem Theorem_7_4_2 {m a : Nat} [NeZero m] (h1 : rel_prime m a) :
  [a]_m ^ (phi m) = [1]_m := by
  have h2 : invertible (prod_seq m 0 (F m)) := Fprod_invertible m m
  obtain (Y : ZMod m) (h3 : prod_seq m 0 (F m) * Y = [1]_m) from h2
  show [a]_m ^ (phi m) = [1]_m from
    calc [a]_m ^ (phi m)
      _ = [a]_m ^ (phi m) * [1]_m := (Theorem_7_3_6_7 _).symm
      _ = [a]_m ^ (phi m) * (prod_seq m 0 (F m) * Y) := by rw [h3]
      _ = ([a]_m ^ (phi m) * prod_seq m 0 (F m)) * Y := by ring
      _ = prod_seq m 0 (F m o G m a) * Y := by rw [FG_prod h1 m, phi_def]
      _ = prod_seq m 0 (F m) * Y := by
        rw [perm_prod (F m) m (G m a) (G_perm_below h1)]
      _ = [1]_m := by rw [h3]
done
```

Rephrasing this theorem in terms of numbers gives us the usual statement of Euler's theorem:

```
lemma Exercise_7_4_5_Int (m : Nat) (a : Int) :
  ∀ (n : Nat), [a]_m ^ n = [a ^ n]_m := sorry

lemma Exercise_7_4_5_Nat (m a n : Nat) :
  [a]_m ^ n = [a ^ n]_m := by
  rewrite [Exercise_7_4_5_Int]
  rfl
  done

theorem Euler's_theorem {m a : Nat} [NeZero m]
  (h1 : rel_prime m a) : a ^ (phi m) ≡ 1 (MOD m) := by
  have h2 : [a]_m ^ (phi m) = [1]_m := Theorem_7_4_2 h1
  rewrite [Exercise_7_4_5_Nat m a (phi m)] at h2
  --h2 : [a ^ phi m]_m = [1]_m
  show a ^ (phi m) ≡ 1 (MOD m) from (cc_eq_iff_congr _ _ _).ltr h2
  done

#eval gcd 10 7      --Answer: 1. So 10 and 7 are relatively prime

#eval 7 ^ phi 10    --Answer: 2401, which is congruent to 1 mod 10.
```

We now turn to the two lemmas whose proofs we skipped over, `G_perm_below` and `perm_prod`. To prove `G_perm_below`, we must prove three facts: `maps_below m (G m a)`, `one_one_below m (G m a)`, and `onto_below m (G m a)`. The first is straightforward:

```
lemma G_maps_below (m a : Nat) [NeZero m] : maps_below m (G m a) := by
  define          --Goal : ∀ i < m, G m a i < m
  fix i : Nat
  assume h1 : i < m
  rewrite [G_def]    --Goal : a * i % m < m
  show a * i % m < m from mod_nonzero_lt (a * i) (NeZero.ne m)
  done
```

For the second and third, we start with lemmas that are reminiscent of Theorem 5.3.3.

```
lemma right_inv_onto_below {n : Nat} {g g' : Nat → Nat}
  (h1 : ∀ i < n, g (g' i) = i) (h2 : maps_below n g') :
  onto_below n g := by
  define at h2; define
  fix k : Nat
  assume h3 : k < n
  apply Exists.intro (g' k)
  show g' k < n ∧ g (g' k) = k from And.intro (h2 k h3) (h1 k h3)
  done

lemma left_inv_one_one_below {n : Nat} {g g' : Nat → Nat}
  (h1 : ∀ i < n, g' (g i) = i) : one_one_below n g := sorry
```

To apply these lemmas with $g = G m a$, we need a function to play the role of g' . A natural choice is $G m a'$, where a' is chosen so that $[a']_m$ is the multiplicative inverse of $[a]_m$. We know from earlier work that if m and a are relatively prime then the multiplicative inverse of $[a]_m$ is $[\text{gcd_c2 } m \ a]_m$. However, in the notation $G m a'$, we can't let $a' = \text{gcd_c2 } m \ a$, because a' must be a natural number and $\text{gcd_c2 } a$ is an integer. And we can't simply coerce an integer to be a natural number—what if it's negative? But we know $[\text{gcd_c2 } m \ a]_m = [(\text{gcd_c2 } m \ a) \% m]_m$ and $0 \leq (\text{gcd_c2 } m \ a) \% m$, and there is a function, `Int.toNat`, that will convert a nonnegative integer to a natural number. So we make the following definitions:

```
def inv_mod (m a : Nat) : Nat := Int.toNat ((gcd_c2 m a) \% m)

def Ginv (m a i : Nat) : Nat := G m (inv_mod m a) i
```

Now $Ginv m a$ can play the role of g' in the last two lemmas. We'll skip the details and just summarize the results.

```

lemma Ginv_right_inv {m a : Nat} [NeZero m] (h1 : rel_prime m a) :
  ∀ i < m, G m a (Ginv m a i) = i := sorry

lemma Ginv_left_inv {m a : Nat} [NeZero m] (h1 : rel_prime m a) :
  ∀ i < m, Ginv m a (G m a i) = i := sorry

lemma Ginv_maps_below (m a : Nat) [NeZero m] :
  maps_below m (Ginv m a) := G_maps_below m (inv_mod m a)

lemma G_one_one_below {m a : Nat} [NeZero m] (h1 : rel_prime m a) :
  one_one_below m (G m a) :=
  left_inv_one_one_below (Ginv_left_inv h1)

lemma G_onto_below {m a : Nat} [NeZero m] (h1 : rel_prime m a) :
  onto_below m (G m a) :=
  right_inv_onto_below (Ginv_right_inv h1) (Ginv_maps_below m a)

lemma G_perm_below {m a : Nat} [NeZero m] (h1 : rel_prime m a) :
  perm_below m (G m a) := And.intro (G_maps_below m a)
  (And.intro (G_one_one_below h1) (G_onto_below h1))

```

Finally, we turn to the proof of `perm_prod`. Our proof will be by mathematical induction. In the induction step, our induction hypothesis will be

$$\forall (g : \text{Nat} \rightarrow \text{Nat}), \text{perm_below } n \ g \rightarrow \\ \text{prod_seq } n \ 0 \ f = \text{prod_seq } n \ 0 \ (f \circ g),$$

and we will have to prove

$$\forall (g : \text{Nat} \rightarrow \text{Nat}), \text{perm_below } (n + 1) \ g \rightarrow \\ \text{prod_seq } (n + 1) \ 0 \ f = \text{prod_seq } (n + 1) \ 0 \ (f \circ g).$$

To prove this, we'll introduce an arbitrary function $g : \text{Nat} \rightarrow \text{Nat}$ and assume `perm_below (n + 1) g`. How can we make use of the inductive hypothesis? Here's the key idea: Since g permutes the numbers below $n + 1$, there must be some $u \leq n$ such that $g u = n$. Now let s be a function that swaps u and n , but leaves all other numbers fixed. In other words, $s u = n$, $s n = u$, and $s i = i$ if $i \neq u$ and $i \neq n$. It is not hard to show that s permutes the numbers below $n + 1$, and using that fact we can prove that $g \circ s$ permutes the numbers below $n + 1$. But notice that

$$(g \circ s) n = g (s n) = g u = n.$$

7.4. Euler's Theorem

In other words, $g \circ s$ leaves n fixed. Using that fact, we'll be able to prove that $g \circ s$ permutes the numbers below n . We can therefore apply the inductive hypothesis to $g \circ s$, which leads to the conclusion

```
prod_seq n 0 f = prod_seq n 0 (f ∘ g ∘ s).
```

Since we also have $(f ∘ g ∘ s) n = f ((g ∘ s) n) = f n$, we can extend this to

```
prod_seq (n + 1) 0 f = prod_seq (n + 1) 0 (f ∘ g ∘ s).
```

Finally, we can then reach the required conclusion by proving

```
prod_seq (n + 1) 0 (f ∘ g ∘ s) = prod_seq (n + 1) 0 (f ∘ g).
```

To carry out this plan, we begin by defining our “swapping” function and proving its properties.

```
def swap (u v i : Nat) : Nat :=
  if i = u then v else if i = v then u else i

lemma swap_fst (u v : Nat) : swap u v u = v := by
  define : swap u v u
  --Goal : (if u = u then v else if u = v then u else u) = v
  have h : u = u := by rfl
  rewrite [if_pos h]
  rfl
  done

lemma swap_snd (u v : Nat) : swap u v v = u := sorry

lemma swap_perm_below {u v n} (h1 : u < n) (h2 : v < n) :
  perm_below n (swap u v) := sorry
```

For the swapping function s in the proof outline above, we'll use $\text{swap } u \ n$. To prove that $g \circ \text{swap } u \ n$ permutes the numbers below n , we'll need two lemmas:

```
lemma comp_perm_below {n : Nat} {f g : Nat → Nat}
  (h1 : perm_below n f) (h2 : perm_below n g) :
  perm_below n (f ∘ g) := sorry

lemma perm_below_fixed {n : Nat} {g : Nat → Nat}
  (h1 : perm_below (n + 1) g) (h2 : g n = n) : perm_below n g := sorry
```

For the final step of the proof, we'll need several lemmas

```

lemma break_prod_twice {m u j n : Nat} (f : Nat → ZMod m)
  (h1 : n = u + 1 + j) : prod_seq (n + 1) 0 f =
    prod_seq u 0 f * f u * prod_seq j (u + 1) f * f n := sorry

lemma swap_prod_eq_prod_below {m u n : Nat} (f : Nat → ZMod m)
  (h1 : u ≤ n) : prod_seq u 0 (f ∘ swap u n) = prod_seq u 0 f := sorry

lemma swap_prod_eq_prod_between {m u j n : Nat} (f : Nat → ZMod m)
  (h1 : n = u + 1 + j) : prod_seq j (u + 1) (f ∘ swap u n) =
    prod_seq j (u + 1) f := sorry

lemma trivial_swap (u : Nat) : swap u u = id := sorry

```

Using these lemmas, we can prove the fact we'll need in the final step.

```

lemma swap_prod_eq_prod {m u n : Nat} (f : Nat → ZMod m) (h1 : u ≤ n) :
  prod_seq (n + 1) 0 (f ∘ swap u n) = prod_seq (n + 1) 0 f := by
  by_cases h2 : u = n
  . -- Case 1. h2 : u = n
    rewrite [h2, trivial_swap n]
    --Goal : prod_seq (n + 1) 0 (f ∘ id) = prod_seq (n + 1) 0 f
    rfl
    done
  . -- Case 2. h2 : ¬u = n
    have h3 : u + 1 ≤ n := Nat.lt_of_le_of_ne h1 h2
    obtain (j : Nat) (h4 : n = u + 1 + j) from Nat.exists_eq_add_of_le h3
    have break_f : prod_seq (n + 1) 0 f =
      prod_seq u 0 f * f u * prod_seq j (u + 1) f * f n :=
      break_prod_twice f h4
    have break_fs : prod_seq (n + 1) 0 (f ∘ swap u n) =
      prod_seq u 0 (f ∘ swap u n) * (f ∘ swap u n) u *
      prod_seq j (u + 1) (f ∘ swap u n) * (f ∘ swap u n) n :=
      break_prod_twice (f ∘ swap u n) h4
    have f_eq_fs_below : prod_seq u 0 (f ∘ swap u n) =
      prod_seq u 0 f := swap_prod_eq_prod_below f h1
    have f_eq_fs_btwn : prod_seq j (u + 1) (f ∘ swap u n) =
      prod_seq j (u + 1) f := swap_prod_eq_prod_between f h4
    show prod_seq (n + 1) 0 (f ∘ swap u n) = prod_seq (n + 1) 0 f from
      calc prod_seq (n + 1) 0 (f ∘ swap u n) =
        _ = prod_seq u 0 (f ∘ swap u n) * (f ∘ swap u n) u *
          prod_seq j (u + 1) (f ∘ swap u n) * (f ∘ swap u n) n :=
        break_fs

```

```

_ = prod_seq u 0 f * (f ∘ swap u n) u *
prod_seq j (u + 1) f * (f ∘ swap u n) n := by
  rw [f_eq_fs_below, f_eq_fs_btwn]
_ = prod_seq u 0 f * f (swap u n u) *
prod_seq j (u + 1) f * f (swap u n n) := by rfl
_ = prod_seq u 0 f * f n * prod_seq j (u + 1) f * f u := by
  rw [swap_fst, swap_snd]
_ = prod_seq u 0 f * f u * prod_seq j (u + 1) f * f n := by ring
_ = prod_seq (n + 1) 0 f := break_f.symm
done
done

```

We finally have all the pieces in place to prove `perm_prod`:

```

lemma perm_prod {m : Nat} (f : Nat → ZMod m) :
  ∀ (n : Nat), ∀ (g : Nat → Nat), perm_below n g →
    prod_seq n 0 f = prod_seq n 0 (f ∘ g) := by
  by_induc
  · -- Base Case
    fix g : Nat → Nat
    assume h1 : perm_below 0 g
    rewrite [prod_seq_base, prod_seq_base]
    rfl
    done
  · -- Induction Step
    fix n : Nat
    assume ih : ∀ (g : Nat → Nat), perm_below n g →
      prod_seq n 0 f = prod_seq n 0 (f ∘ g)
    fix g : Nat → Nat
    assume g_pb : perm_below (n + 1) g
    define at g_pb
    have g_ob : onto_below (n + 1) g := g_pb.right.right
    define at g_ob
    have h1 : n < n + 1 := by linarith
    obtain (u : Nat) (h2 : u < n + 1 ∧ g u = n) from g_ob n h1
    have s_pb : perm_below (n + 1) (swap u n) :=
      swap_perm_below h2.left h1
    have gs_pb_n1 : perm_below (n + 1) (g ∘ swap u n) :=
      comp_perm_below g_pb s_pb
    have gs_fix_n : (g ∘ swap u n) n = n :=
      calc (g ∘ swap u n) n =
        _ = g (swap u n n) := by rfl

```

```

_ = g u := by rw [swap_snd]
_ = n := h2.right
have gs_pb_n : perm_below n (g ∘ swap u n) :=
  perm_below_fixed gs_pb_n1 gs_fix_n
have gs_prod : prod_seq n 0 f = prod_seq n 0 (f ∘ (g ∘ swap u n)) :=
  ih (g ∘ swap u n) gs_pb_n
have h3 : u ≤ n := by linarith
show prod_seq (n + 1) 0 f = prod_seq (n + 1) 0 (f ∘ g) from
calc prod_seq (n + 1) 0 f
  _ = prod_seq n 0 f * f n := prod_seq_zero_step n f
  _ = prod_seq n 0 (f ∘ (g ∘ swap u n)) *
    f ((g ∘ swap u n) n) := by rw [gs_prod, gs_fix_n]
  _ = prod_seq n 0 (f ∘ g ∘ swap u n) *
    (f ∘ g ∘ swap u n) n := by rfl
  _ = prod_seq (n + 1) 0 (f ∘ g ∘ swap u n) :=
    (prod_seq_zero_step n (f ∘ g ∘ swap u n)).symm
  _ = prod_seq (n + 1) 0 ((f ∘ g) ∘ swap u n) := by rfl
  _ = prod_seq (n + 1) 0 (f ∘ g) := swap_prod_eq_prod (f ∘ g) h3
done
done

```

There is one more theorem that is proven in Section 7.4 of *HTPI*: Theorem 7.4.4, which says that φ is a multiplicative function. The proof requires ideas that we will not develop in Lean until Chapter 8, so we will put off the proof until Section 8.1½.

Exercises

- Hint: Use induction.
--For the base case, compute $[a]_m^0 * [1]_m$ in two ways:
--by Theorem_7_3_6_7, $[a]^0 * [1]_m = [a]_m^0$
--by ring, $[a]_m^0 * [1]_m = [1]_m$.
lemma Exercise_7_4_5_Int (m : Nat) (a : Int) :
 ∀ (n : Nat), [a]_m^n = [a^n]_m := sorry
- lemma left_inv_one_one_below {n : Nat} {g g' : Nat → Nat}
 (h1 : ∀ i < n, g' (g i) = i) : one_one_below n g := sorry
- lemma comp_perm_below {n : Nat} {f g : Nat → Nat}
 (h1 : perm_below n f) (h2 : perm_below n g) :
 perm_below n (f ∘ g) := sorry

```

4. lemma perm_below_fixed {n : Nat} {g : Nat → Nat}
   (h1 : perm_below (n + 1) g) (h2 : g n = n) : perm_below n g := sorry

5. lemma Lemma_7_4_6 {a b c : Nat} :
   rel_prime (a * b) c ↔ rel_prime a c ∧ rel_prime b c := sorry

6. example {m a : Nat} [NeZero m] (h1 : rel_prime m a) :
   a ^ (phi m + 1) ≡ a (MOD m) := sorry

7. theorem Like_Exercise_7_4_11 {m a p : Nat} [NeZero m]
   (h1 : rel_prime m a) (h2 : p + 1 = phi m) :
   [a]_m * [a ^ p]_m = [1]_m := sorry

8. theorem Like_Exercise_7_4_12 {m a p q k : Nat} [NeZero m]
   (h1 : rel_prime m a) (h2 : p = q + (phi m) * k) :
   a ^ p ≡ a ^ q (MOD m) := sorry

```

7.5. Public-Key Cryptography

Section 7.5 of *HTPI* discusses the RSA public-key cryptography system. The system is based on the following theorem:

```

theorem Theorem_7_5_1 (p q n e d k m c : Nat)
  (p_prime : prime p) (q_prime : prime q) (p_ne_q : p ≠ q)
  (n_pq : n = p * q) (ed_congr_1 : e * d = k * (p - 1) * (q - 1) + 1)
  (h1 : [m]_n ^ e = [c]_n) : [c]_n ^ d = [m]_n

```

For an explanation of how the RSA system works and why `Theorem_7_5_1` justifies it, see *HTPI*. Here we will focus on proving the theorem in Lean.

We will be applying Euler's theorem to the prime numbers p and q , so we will need to know how to compute $\phi(p)$ and $\phi(q)$. Fortunately, there is a simple formula we can use.

```

lemma num_rp_prime {p : Nat} (h1 : prime p) :
  ∀ k < p, num_rp_below p (k + 1) = k := sorry

lemma phi_prime {p : Nat} (h1 : prime p) : phi p = p - 1 := by
  have h2 : 1 ≤ p := prime_pos h1
  have h3 : p - 1 + 1 = p := Nat.sub_add_cancel h2
  have h4 : p - 1 < p := by linarith
  have h5 : num_rp_below p (p - 1 + 1) = p - 1 :=
    num_rp_prime h1 (p - 1) h4

```

```
rewrite [h3] at h5
show phi p = p - 1 from h5
done
```

We will also need to use Lemma 7.4.5 from *HTPI*. To prove that lemma in Lean, we will use `Theorem_7_2_2`, which says that for natural numbers a , b , and c , if $c \mid a * b$ and c and a are relatively prime, then $c \mid b$. But we will need to extend the theorem to allow b to be an integer rather than a natural number. To prove this extension, we use the Lean function `Int.natAbs : Int → Nat`, which computes the absolute value of an integer. Lean knows several theorems about this function:

```
@Int.natCast_dvd : ∀ {n : ℤ} {m : ℕ}, ↑m ∣ n ↔ m ∣ Int.natAbs n

Int.natAbs_mul : ∀ (a b : ℤ),
  Int.natAbs (a * b) = Int.natAbs a * Int.natAbs b

Int.natAbs_ofNat : ∀ (n : ℕ), Int.natAbs ↑n = n
```

With the help of these theorems, our extended version of `Theorem_7_2_2` follows easily from the original version:

```
theorem Theorem_7_2_2_Int {a c : Nat} {b : Int}
  (h1 : ↑c ∣ ↑a * b) (h2 : rel_prime a c) : ↑c ∣ b := by
  rewrite [Int.natCast_dvd, Int.natAbs_mul,
    Int.natAbs_ofNat] at h1           --h1 : c ∣ a * Int.natAbs b
  rewrite [Int.natCast_dvd]          --Goal : c ∣ Int.natAbs b
  show c ∣ Int.natAbs b from Theorem_7_2_2 h1 h2
  done
```

With that preparation, we can now prove `Lemma_7_4_5`.

```
lemma Lemma_7_4_5 {m n : Nat} (a b : Int) (h1 : rel_prime m n) :
  a ≡ b (MOD m * n) ↔ a ≡ b (MOD m) ∧ a ≡ b (MOD n) := by
  apply Iff.intro
  · -- (→)
    assume h2 : a ≡ b (MOD m * n)
    obtain (j : Int) (h3 : a - b = (m * n) * j) from h2
    apply And.intro
    · -- Proof of a ≡ b (MOD m)
      apply Exists.intro (n * j)
      show a - b = m * (n * j) from
        calc a - b = _
```

```

_ = m * n * j := h3
_ = m * (n * j) := by ring
done
· -- Proof of a ≡ b (MOD n)
apply Exists.intro (m * j)
show a - b = n * (m * j) from
calc a - b
_ = m * n * j := h3
_ = n * (m * j) := by ring
done
done
· -- (↔)
assume h2 : a ≡ b (MOD m) ∧ a ≡ b (MOD n)
obtain (j : Int) (h3 : a - b = m * j) from h2.left
have h4 : (↑n : Int) | a - b := h2.right
rewrite [h3] at h4      --h4 : ↑n | ↑m * j
have h5 : ↑n | j := Theorem_7_2_2_Int h4 h1
obtain (k : Int) (h6 : j = n * k) from h5
apply Exists.intro k    --Goal : a - b = ↑(m * n) * k
rewrite [Nat.cast_mul]  --Goal : a - b = ↑m * ↑n * k
show a - b = (m * n) * k from
calc a - b
_ = m * j := h3
_ = m * (n * k) := by rw [h6]
_ = (m * n) * k := by ring
done
done

```

Finally, we will need an exercise from Section 7.2, and we will need to know `NeZero p` for prime numbers p :

```

theorem rel_prime_symm {a b : Nat} (h : rel_prime a b) :
rel_prime b a := sorry

lemma prime_NeZero {p : Nat} (h : prime p) : NeZero p := by
rewrite [neZero_iff]      --Goal : p ≠ 0
define at h
linarith
done

```

Much of the reasoning about modular arithmetic that we need for the proof of `Theorem_7_5_1` is contained in a technical lemma:

```

lemma Lemma_7_5_1 {p e d m c s : Nat} {t : Int}
  (h1 : prime p) (h2 : e * d = (p - 1) * s + 1)
  (h3 : m ^ e - c = p * t) :
  c ^ d ≡ m (MOD p) := by
  have h4 : m ^ e ≡ c (MOD p) := Exists.intro t h3
  have h5 : [m ^ e]_p = [c]_p := (cc_eq_iff_congr _ _ _).rtl h4
  rewrite [←Exercise_7_4_5_Nat] at h5 --h5 : [m]_p ^ e = [c]_p
  by_cases h6 : p | m
  · -- Case 1. h6 : p | m
    have h7 : m ≡ 0 (MOD p) := by
    obtain (j : Nat) (h8 : m = p * j) from h6
    apply Exists.intro (↑j : Int) --Goal : ↑m - 0 = ↑p * ↑j
    rewrite [h8, Nat.cast_mul]
    ring
    done
    have h8 : [m]_p = [0]_p := (cc_eq_iff_congr _ _ _).rtl h7
    have h9 : e * d ≠ 0 := by
      rewrite [h2]
      show (p - 1) * s + 1 ≠ 0 from Nat.add_one_ne_zero _
      done
    have h10 : (0 : Int) ^ (e * d) = 0 := zero_pow h9
    have h11 : [c ^ d]_p = [m]_p :=
      calc [c ^ d]_p
        _ = [c]_p ^ d := by rw [Exercise_7_4_5_Nat]
        _ = ([m]_p ^ e) ^ d := by rw [h5]
        _ = [m]_p ^ (e * d) := by ring
        _ = [0]_p ^ (e * d) := by rw [h8]
        _ = [0 ^ (e * d)]_p := Exercise_7_4_5_Int _ _ _
        _ = [0]_p := by rw [h10]
        _ = [m]_p := by rw [h8]
    show c ^ d ≡ m (MOD p) from (cc_eq_iff_congr _ _ _).ltr h11
    done
  · -- Case 2. h6 : ¬p | m
    have h7 : rel_prime m p := rel_prime_of_prime_not_dvd h1 h6
    have h8 : rel_prime p m := rel_prime_symm h7
    have h9 : NeZero p := prime_NeZero h1
    have h10 : (1 : Int) ^ s = 1 := by ring
    have h11 : [c ^ d]_p = [m]_p :=
      calc [c ^ d]_p
        _ = [c]_p ^ d := by rw [Exercise_7_4_5_Nat]
        _ = ([m]_p ^ e) ^ d := by rw [h5]
        _ = [m]_p ^ (e * d) := by ring

```

```

_ = [m]_p ^ ((p - 1) * s + 1) := by rw [h2]
_ = ([m]_p ^ (p - 1)) ^ s * [m]_p := by ring
_ = ([m]_p ^ (phi p)) ^ s * [m]_p := by rw [phi_prime h1]
_ = [1]_p ^ s * [m]_p := by rw [Theorem_7_4_2 h8]
_ = [1 ^ s]_p * [m]_p := by rw [Exercise_7_4_5_Int]
_ = [1]_p * [m]_p := by rw [h10]
_ = [m]_p * [1]_p := by ring
_ = [m]_p := Theorem_7_3_6_7 _

show c ^ d ≡ m (MOD p) from (cc_eq_iff_congr _ _ _).ltr h11
done
done

```

Here, finally, is the proof of Theorem_7_5_1:

```

theorem Theorem_7_5_1 (p q n e d k m c : Nat)
  (p_prime : prime p) (q_prime : prime q) (p_ne_q : p ≠ q)
  (n_pq : n = p * q) (ed_congr_1 : e * d = k * (p - 1) * (q - 1) + 1)
  (h1 : [m]_n ^ e = [c]_n) : [c]_n ^ d = [m]_n := by
  rewrite [Exercise_7_4_5_Nat, cc_eq_iff_congr] at h1
  --h1 : m ^ e ≡ c (MOD n)
  rewrite [Exercise_7_4_5_Nat, cc_eq_iff_congr]
  --Goal : c ^ d ≡ m (MOD n)
  obtain (j : Int) (h2 : m ^ e - c = n * j) from h1
  rewrite [n_pq, Nat.cast_mul] at h2
  --h2 : m ^ e - c = p * q * j
  have h3 : e * d = (p - 1) * (k * (q - 1)) + 1 := by
    rewrite [ed_congr_1]
    ring
    done
  have h4 : m ^ e - c = p * (q * j) := by
    rewrite [h2]
    ring
    done
  have congr_p : c ^ d ≡ m (MOD p) := Lemma_7_5_1 p_prime h3 h4
  have h5 : e * d = (q - 1) * (k * (p - 1)) + 1 := by
    rewrite [ed_congr_1]
    ring
    done
  have h6 : m ^ e - c = q * (p * j) := by
    rewrite [h2]
    ring
    done

```

```

have congr_q : c ^ d ≡ m (MOD q) := Lemma_7_5_1 q_prime h5 h6
have h7 : ¬q | p := by
  by_contra h8
  have h9 : q = 1 ∨ q = p := dvd_prime p_prime h8
  disj_syll h9 (prime_not_one q_prime)
  show False from p_neq h9.symm
  done
have h8 : rel_prime p q := rel_prime_of_prime_not_dvd q_prime h7
rewrite [n_pq, Lemma_7_4_5 _ _ h8]
show c ^ d ≡ m (MOD p) ∧ c ^ d ≡ m (MOD q) from
  And.intro congr_p congr_q
done

```

Exercises

1. --Hint: Use induction.
`lemma num_rp_prime {p : Nat} (h1 : prime p) :
 ∀ k < p, num_rp_below p (k + 1) = k := sorry`
2. `lemma three_prime : prime 3 := sorry`
3. --Hint: Use the previous exercise, Exercise_7_2_7, and Theorem_7_4_2.
`theorem Exercise_7_5_13a (a : Nat) (h1 : rel_prime 561 a) :
 a ^ 560 ≡ 1 (MOD 3) := sorry`
4. --Hint: Imitate the way Theorem_7_2_2_Int was proven from Theorem_7_2_2.
`lemma Theorem_7_2_3_Int {p : Nat} {a b : Int}
 (h1 : prime p) (h2 : ↑p | a * b) : ↑p | a ∨ ↑p | b := sorry`
5. --Hint: Use the previous exercise.
`theorem Exercise_7_5_14b (n : Nat) (b : Int)
 (h1 : prime n) (h2 : b ^ 2 ≡ 1 (MOD n)) :
 b ≡ 1 (MOD n) ∨ b ≡ -1 (MOD n) := sorry`

8 Infinite Sets

8.1. Equinumerous Sets

Chapter 8 of *HTPI* begins by defining a set A to be *equinumerous* with a set B if there is a function $f : A \rightarrow B$ that is one-to-one and onto. But in Lean, a function must go from a *type* to a *type*, not a set to a set. Thus, when we translate the *HTPI* definition into Lean's language, we end up with a definition that tells us when one *type* is equinumerous with another. Throughout this chapter, we will use the letters U, V, \dots for types and A, B, \dots for sets, so we state the definition of *equinumerous* like this in Lean:

```
def equinum (U V : Type) : Prop :=  
  ∃ (f : U → V), one_to_one f ∧ onto f
```

As in *HTPI*, we introduce the notation $U \sim V$ to indicate that U is equinumerous with V (to enter the symbol \sim , type `\sim` or `\sim`).

```
notation:50  U:50 " ~ " V:50 => equinum U V
```

Section 8.1 of *HTPI* begins the study of this concept with some examples. The first is a one-to-one, onto function from \mathbb{Z}^+ to \mathbb{Z} , which shows that $\mathbb{Z}^+ \sim \mathbb{Z}$. We will modify this example slightly to make it a function `fnz` from `Nat` to `Int`:

```
def fnz (n : Nat) : Int := if 2 | n then ↑(n / 2) else -↑((n + 1) / 2)
```

Note that, to get a result of type `Int`, coercion is necessary. We have specified that the coercion should be done after the computation of either $n / 2$ or $(n + 1) / 2$, with that computation being done using natural-number arithmetic. Checking a few values of this functions suggests a simple pattern:

```
#eval [fnz 0, fnz 1, fnz 2, fnz 3, fnz 4, fnz 5, fnz 6]  
--Answer: [0, -1, 1, -2, 2, -3, 3]
```

Perhaps the easiest way to prove that `fnz` is one-to-one and onto is to define a function that turns out to be its inverse. This time, in order to get the right type for the value of the function, we use the function `Int.toNat` to convert a nonnegative integer to a natural number.

8.1. Equinumerous Sets

```
def fnz (a : Int) : Nat :=
  if a ≥ 0 then 2 * Int.toNat a else 2 * Int.toNat (-a) - 1

#eval [fnz 0, fnz (-1), fnz 1, fnz (-2), fnz 2, fnz (-3), fnz 3]
--Answer: [0, 1, 2, 3, 4, 5, 6]
```

To prove that fnz is the inverse of fnz , we begin by proving lemmas making it easier to compute the values of these functions

```
lemma fnz_even (k : Nat) : fnz (2 * k) = ↑k := by
  have h1 : 2 | 2 * k := by
    apply Exists.intro k
    rfl
    done
  have h2 : 0 < 2 := by linarith
  show fnz (2 * k) = ↑k from
    calc fnz (2 * k)
      _ = ↑(2 * k / 2) := if_pos h1
      _ = ↑k := by rw [Nat.mul_div_cancel_left k h2]
  done

lemma fnz_odd (k : Nat) : fnz (2 * k + 1) = -↑(k + 1) := sorry

lemma fnz_nat (k : Nat) : fnz ↑k = 2 * k := by rfl

lemma fnz_neg_succ_nat (k : Nat) : fnz (-↑(k + 1)) = 2 * k + 1 := by rfl
```

Using these lemmas and reasoning by cases, it is straightforward to prove lemmas confirming that the composition of these functions, in either order, yields the identity function. The cases for the first lemma are based on an exercise from Section 6.1.

```
lemma fnz_fnz : fnz ∘ fnz = id := by
  apply funext          --Goal : ∀ (x : Nat), (fnz ∘ fnz) x = id x
  fix n : Nat
  rewrite [comp_def]   --Goal : fnz (fnz n) = id n
  have h1 : nat_even n ∨ nat_odd n := Exercise_6_1_16a1 n
  by_cases on h1
  · -- Case 1. h1 : nat_even n
    obtain (k : Nat) (h2 : n = 2 * k) from h1
    rewrite [h2, fnz_even, fnz_nat]
    rfl
```

8.1. Equinumerous Sets

```

done
· -- Case 2. h1 : nat_odd n
obtain (k : Nat) (h2 : n = 2 * k + 1) from h1
  rewrite [h2, fnz_odd, fnz_neg_succ_nat]
rfl
done
done

lemma fnz_fnz : fnz ∘ fnz = id := sorry

```

By theorems from Chapter 5, it follows that both `fnz` and `fzn` are one-to-one and onto.

```

lemma fzn_one_one : one_to_one fzn := Theorem_5_3_3_1 fzn fnz fnz_fnz

lemma fzn_onto : onto fzn := Theorem_5_3_3_2 fzn fnz fzn_fnz

lemma fnz_one_one : one_to_one fnz := Theorem_5_3_3_1 fnz fzn fzn_fnz

lemma fnz_onto : onto fnz := Theorem_5_3_3_2 fnz fzn fnz_fnz

```

We conclude that `Nat ~ Int` and `Int ~ Nat`:

```

theorem N_equinum_Z : Nat ~ Int :=
  Exists.intro fnz (And.intro fnz_one_one fnz_onto)

theorem Z_equinum_N : Int ~ Nat :=
  Exists.intro fzn (And.intro fzn_one_one fzn_onto)

```

We'll give one more example: a one-to-one, onto function `fnnn` from `Nat × Nat` to `Nat`, whose definition is modeled on a function from $\mathbb{Z}^+ \times \mathbb{Z}^+$ to \mathbb{Z}^+ in *HTPI*. The definition of `fnnn` will use numbers of the form $k * (k + 1) / 2$. These numbers are sometimes called *triangular numbers*, because they count the number of objects in a triangular grid with k rows.

```

def tri (k : Nat) : Nat := k * (k + 1) / 2

def fnnn (p : Nat × Nat) : Nat := tri (p.1 + p.2) + p.1

lemma fnnn_def (a b : Nat) : fnnn (a, b) = tri (a + b) + a := by rfl

#eval [fnnn (0, 0), fnnn (0, 1), fnnn (1, 0),
      fnnn (0, 2), fnnn (1, 1), fnnn (2, 0)]
--Answer: [0, 1, 2, 3, 4, 5]

```

8.1. Equinumerous Sets

Two simple lemmas about `tri`, whose proofs we leave as exercises for you, help us prove the important properties of `fnnn`:

```

lemma tri_step (k : Nat) : tri (k + 1) = tri k + k + 1 := sorry

lemma tri_incr {j k : Nat} (h1 : j ≤ k) : tri j ≤ tri k := sorry

lemma le_of_fnnn_eq {a1 b1 a2 b2 : Nat}
  (h1 : fnnn (a1, b1) = fnnn (a2, b2)) : a1 + b1 ≤ a2 + b2 := by
  by_contra h2
  have h3 : a2 + b2 + 1 ≤ a1 + b1 := by linarith
  have h4 : fnnn (a2, b2) < fnnn (a1, b1) :=
    calc fnnn (a2, b2)
      _ = tri (a2 + b2) + a2 := by rfl
      _ < tri (a2 + b2) + (a2 + b2) + 1 := by linarith
      _ = tri (a2 + b2 + 1) := (tri_step _).symm
      _ ≤ tri (a1 + b1) := tri_incr h3
      _ ≤ tri (a1 + b1) + a1 := by linarith
      _ = fnnn (a1, b1) := by rfl
  linarith
  done

lemma fnnn_one_one : one_to_one fnnn := by
  fix (a1, b1) : Nat × Nat
  fix (a2, b2) : Nat × Nat
  assume h1 : fnnn (a1, b1) = fnnn (a2, b2) --Goal : (a1, b1) = (a2, b2)
  have h2 : a1 + b1 ≤ a2 + b2 := le_of_fnnn_eq h1
  have h3 : a2 + b2 ≤ a1 + b1 := le_of_fnnn_eq h1.symm
  have h4 : a1 + b1 = a2 + b2 := by linarith
  rewrite [fnnn_def, fnnn_def, h4] at h1
  --h1 : tri (a2 + b2) + a1 = tri (a2 + b2) + a2
  have h6 : a1 = a2 := Nat.add_left_cancel h1
  rewrite [h6] at h4 --h4 : a2 + b1 = a2 + b2
  have h7 : b1 = b2 := Nat.add_left_cancel h4
  rewrite [h6, h7]
  rfl
  done

lemma fnnn_onto : onto fnnn := by
  define --Goal : ∀ (y : Nat), ∃ (x : Nat × Nat), fnnn x = y
  by_induc
  · -- Base Case
    apply Exists.intro (0, 0)

```

```
rfl
done
· -- Induction Step
fix n : Nat
assume ih : ∃ (x : Nat × Nat), fnnn x = n
obtain ((a, b) : Nat × Nat) (h1 : fnnn (a, b) = n) from ih
by_cases h2 : b = 0
· -- Case 1. h2 : b = 0
apply Exists.intro (0, a + 1)
show fnnn (0, a + 1) = n + 1 from
calc fnnn (0, a + 1)
_ = tri (0 + (a + 1)) + 0 := by rfl
_ = tri (a + 1) := by ring
_ = tri a + a + 1 := tri_step a
_ = tri (a + 0) + a + 1 := by ring
_ = fnnn (a, b) + 1 := by rw [h2, fnnn_def]
_ = n + 1 := by rw [h1]
done
· -- Case 2. h2 : b ≠ 0
obtain (k : Nat) (h3 : b = k + 1) from
exists_eq_add_one_of_ne_zero h2
apply Exists.intro (a + 1, k)
show fnnn (a + 1, k) = n + 1 from
calc fnnn (a + 1, k)
_ = tri (a + 1 + k) + (a + 1) := by rfl
_ = tri (a + (k + 1)) + a + 1 := by ring
_ = tri (a + b) + a + 1 := by rw [h3]
_ = fnnn (a, b) + 1 := by rfl
_ = n + 1 := by rw [h1]
done
done
done

theorem NxN_equinum_N : (Nat × Nat) ~ Nat :=
Exists.intro fnnn (And.intro fnnn_one_one fnnn_onto)
```

One of the most important theorems about the concept of equinumerosity is Theorem 8.1.3 in *HTPI*, which says that \sim is reflexive, symmetric, and transitive. We'll prove the three parts of this theorem separately. To prove that \sim is reflexive, we use the identity function. (Recall from Section 5.1 that $@id U$ is the identity function on the type U .)

8.1. Equinumerous Sets

```

lemma id_one_one (U : Type) : one_to_one (@id U) := by
  fix x1 : U; fix x2 : U
  assume h : id x1 = id x2
  show x1 = x2 from h
  done

lemma id_onto (U : Type) : onto (@id U) := by
  fix y : U           --Goal : ∃ (x : U), id x = y
  apply Exists.intro y --Goal : id y = y
  rfl
  done

theorem Theorem_8_1_3_1 (U : Type) : U ~ U := by
  apply Exists.intro id
  show one_to_one id ∧ onto id from And.intro (id_one_one U) (id_onto U)
  done

```

For symmetry, we use some theorems from Chapter 5 about inverses of functions:

```

theorem Theorem_8_1_3_2 {U V : Type}
  (h : U ~ V) : V ~ U := by
  obtain (f : U → V) (h1 : one_to_one f ∧ onto f) from h
  obtain (finv : V → U) (h2 : graph finv = inv (graph f)) from
    Theorem_5_3_1 f h1.left h1.right
  apply Exists.intro finv
  have h3 : finv ∘ f = id := Theorem_5_3_2_1 f finv h2
  have h4 : f ∘ finv = id := Theorem_5_3_2_2 f finv h2
  show one_to_one finv ∧ onto finv from
    And.intro (Theorem_5_3_3_1 finv f h4) (Theorem_5_3_3_2 finv f h3)
  done

```

Finally, for transitivity, we use theorems about composition of functions:

```

theorem Theorem_8_1_3_3 {U V W : Type}
  (h1 : U ~ V) (h2 : V ~ W) : U ~ W := by
  obtain (f : U → V) (h3 : one_to_one f ∧ onto f) from h1
  obtain (g : V → W) (h4 : one_to_one g ∧ onto g) from h2
  apply Exists.intro (g ∘ f)
  show one_to_one (g ∘ f) ∧ onto (g ∘ f) from
    And.intro (Theorem_5_2_5_1 f g h3.left h4.left)
    (Theorem_5_2_5_2 f g h3.right h4.right)
  done

```

8.1. Equinumerous Sets

So far, we have only talked about *types* being equinumerous, but later in this chapter we are going to want to talk about *sets* being equinumerous. For example, it would be nice if we could give this proof:

```
theorem wishful_thinking?
{U : Type} (A : Set U) : A ~ A := Theorem_8_1_3_1 A
```

It seems like Lean shouldn't accept this theorem; the notation \sim was defined to apply to types, and in this theorem, A is a set, not a type. But if you enter this theorem into Lean, you will find that Lean accepts it! How is that possible?

We can find out what Lean thinks this theorem means by giving the command `#check @wishful_thinking?`. Lean's response is:

```
@wishful_thinking? : ∀ {U : Type} (A : Set U), ↑A ~ ↑A
```

Aha! The uparrows are the key to unlocking the mystery. As we know, uparrows in Lean represent coercions. So Lean must have coerced A into a type, so that it can be used with the \sim notation. Although A is a set, $\uparrow A$ is a type.

What is the type $\uparrow A$? Intuitively, you can think of the objects of type $\uparrow A$ as the elements of A . Since A has type `Set U`, the elements of A are some, but perhaps not all, of the objects of type U ; for this reason, $\uparrow A$ is called a *subtype* of U .

However, this intuitive description can be misleading. The relationship between $\uparrow A$ and U is actually similar to the relationship between `Nat` and `Int`. Recall that, although we think of the natural numbers as being contained in the integers, in Lean, the types `Nat` and `Int` are completely separate. If n has type `Nat`, then n is not an integer, but there is an integer that corresponds to n , and n can be coerced to that corresponding integer. Similarly, although we might think of $\uparrow A$ as being contained in U , in fact the two types are completely separate. If a has type $\uparrow A$, then a does not have type U , but there is an object of type U that corresponds to a . That corresponding object is called the *value* of a , and it is denoted `a.val`. Furthermore, a can be coerced to `a.val`; using Lean's notation for coercion, we can write `↑a = a.val`.

If a has type $\uparrow A$, then not only is `a.val` an object of type U , but it is an element of A . Indeed, a supplies us with a proof of this fact. This proof is denoted `a.property`. In other words, we have `a.property : a.val ∈ A`. Indeed, you might think of any object $a : \uparrow A$ as a bundle consisting of two pieces of data, `a.val` and `a.property`. Not only can we extract these two pieces of data from a by using the `.val` and `.property` notation, but we can go in the other direction. That is, if we have $x : U$ and $h : x \in A$, then we can bundle these two pieces of data together to create an object of type $\uparrow A$. This object of type $\uparrow A$ is denoted `Subtype.mk x h`. Thus, if $a = Subtype.mk x h$, then a has type $\uparrow A$, $a.val = x$, and $a.property = h$. We can make the creation of objects of type $\uparrow A$ slightly simpler by introducing the following function:

```
def Subtype_elt {U : Type} {A : Set U} {x : U} (h : x ∈ A) : ↑A :=
  Subtype.mk x h
```

Note that the only nonimplicit argument of `Subtype_elt` is `h`. Thus, if we have `h : x ∈ A`, then `Subtype_elt h` is an object of type `↑A` whose value is `x`.

There is one more important property of the type `↑A`. For each element of `A`, there is *only one* corresponding object of type `↑A`. That means that if `a1` and `a2` are two objects of type `↑A` and `a1.val = a2.val`, then `a1 = a2`. We can think of this as an extensionality principle for subtypes. Recall that the extensionality principle for sets is called `Set.ext`, and it says that if two sets have the same elements, then they are equal. Similarly, the extensionality principle for subtypes is denoted `Subtype.ext`, and it says that if two objects of type `↑A` have the same value, then they are equal. More precisely, if we have `a1 a2 : ↑A`, then `Subtype.ext` proves `a1.val = a2.val → a1 = a2`. And just as we usually start a proof that two sets are equal with the tactic `apply Set.ext`, it is often useful to start a proof that two objects of type `↑A` are equal with the tactic `apply Subtype.ext`.

Now that we know that the concept of equinumerosity can be applied not only to types but also to sets, we can use this idea to make a number of definitions. For any natural number n , *HTPI* defines I_n to be the set $\{1, 2, \dots, n\}$, and then it defines a set to be *finite* if it is equinumerous with I_n , for some n . In Lean, it is a bit more convenient to use sets of the form $\{0, 1, \dots, n - 1\}$. With that small change, we can repeat the definitions of finite, denumerable, and countable in *HTPI*.

```
def I (n : Nat) : Set Nat := {k : Nat | k < n}

lemma I_def (k n : Nat) : k ∈ I n ↔ k < n := by rfl

def finite (U : Type) : Prop := ∃ (n : Nat), I n ~ U

def denum (U : Type) : Prop := Nat ~ U

lemma denum_def (U : Type) : denum U ↔ Nat ~ U := by rfl

def ctble (U : Type) : Prop := finite U ∨ denum U
```

Note that in the definition of `finite`, `I n ~ U` means $\uparrow(I n) \sim U$, because `I n` is a set, not a type. But we will usually leave it to Lean to fill in such coercions when necessary.

Theorem 8.1.5 in *HTPI* gives two useful ways to characterize countable sets. The proof of the theorem in *HTPI* uses the fact that every set of natural numbers is countable. *HTPI* gives an intuitive explanation of why this is true, but of course in Lean an intuitive explanation won't

8.1. Equinumerous Sets

do. So before proving a version of Theorem 8.1.5, we sketch a proof that every set of natural numbers is countable.

Suppose A has type `Set Nat`. To prove that A is countable, we will define a function that numbers the elements of A by assigning the number 0 to the smallest element of A , 1 to the next element of A , 2 to the next, and so on. How do we tell which natural number should be assigned to any element m of A ? Notice that if m is the smallest element of A , then there are 0 elements of A that are smaller than m ; if it is the second smallest element of A , then there is 1 element of A that is smaller than m ; and so on. In general, the number assigned to m should be the number of elements of A that are smaller than m . We therefore begin by defining a function `num_elts_below A m` that counts the number of elements of A that are smaller than any natural number m .

The definition of `num_elts_below` is recursive. The recursive step relates the number of elements of A below $n + 1$ to the number of elements below n . There are two possibilities: either $n \in A$ and the number of elements below $n + 1$ is one larger than the number below n , or $n \notin A$ and the two numbers are the same. (This may remind you of the recursion we used to define `num_rp_below` in Chapter 7.)

```
def num_elts_below (A : Set Nat) (m : Nat) : Nat :=
  match m with
  | 0 => 0
  | n + 1 => if n ∈ A then (num_elts_below A n) + 1
               else num_elts_below A n
```

Unfortunately, this definition results in an error message: `failed to synthesize Decidable (n ∈ A)`. Lean is complaining because it doesn't know how to decide, in general, whether or not $n \in A$. As a result, if we asked it to evaluate `num_elts_below A m`, for some particular set A and natural number m , it wouldn't know how to compute it. But this won't be an issue for us; we want to use `num_elts_below` to prove theorems, but we're never going to ask Lean to compute it. For reasons that we won't explain here, we can get Lean to ignore this issue by adding the line `open Classical` at the top of our Lean file. This change leads to a new error, but this time the message is more helpful: `failed to compile definition, consider marking it as 'noncomputable'`. Following Lean's advice, we finally get a definition that is acceptable to Lean:

```
open Classical

noncomputable def num_elts_below (A : Set Nat) (m : Nat) : Nat :=
  match m with
  | 0 => 0
  | n + 1 => if n ∈ A then (num_elts_below A n) + 1
               else num_elts_below A n
```

8.1. Equinumerous Sets

For example, if $A = \{1, 3, 4\}$, then $\text{num_elts_below } A \ 0 = \text{num_elts_below } A \ 1 = 0$, $\text{num_elts_below } A \ 2 = \text{num_elts_below } A \ 3 = 1$, $\text{num_elts_below } A \ 4 = 2$, and $\text{num_elt_below } A \ m = 3$ for every $m \geq 5$. Notice that $\text{num_elts_below } A$ is a function from Nat to Nat , but it is neither one-to-one nor onto. To make it useful for proving that A is countable, we'll need to modify it.

Suppose $f : U \rightarrow V$, but f is not one-to-one or onto. How can we modify f to get a function that is one-to-one or onto? We begin with the problem of getting a function that is onto. The *range* of f is the set of all $y : V$ such that for some $x : U$, $f x = y$:

```
def range {U V : Type} (f : U → V) : Set V := {y : V | ∃ (x : U), f x = y}
```

(In the exercises, we ask you to show that this is the same as $\text{Ran}(\text{graph } f)$.) Since every value of f is in $\text{range } f$, we can convert f into a function from U to $\text{range } f$ (that is, from U to $\uparrow(\text{range } f)$), as follows:

```
lemma elt_range {U V : Type} (f : U → V) (x : U) : f x ∈ range f := by
  define                                     --Goal : ∃ (x_1 : U), f x_1 = f x
  apply Exists.intro x
  rfl
  done

def func_to_range {U V : Type} (f : U → V) (x : U) : range f :=
  Subtype_elt (elt_range f x)
```

According to these definitions, $\text{func_to_range } f$ is a function from U to $\uparrow(\text{range } f)$; it is the same as f , except that each value of the function is converted to an object of type $\uparrow(\text{range } f)$. And it is not hard to prove that this function is onto:

```
lemma ftr_def {U V : Type} (f : U → V) (x : U) :
  (func_to_range f x).val = f x := rfl

lemma ftr_onto {U V : Type} (f : U → V) : onto (func_to_range f) := by
  fix y : range f                      --y has type ↑(range f)
  have h1 : y.val ∈ range f := y.property
  define at h1                         --h1 : ∃ (x : U), f x = y.val
  obtain (x : U) (h2 : f x = y.val) from h1
  apply Exists.intro x                  --Goal : func_to_range f x = y
  apply Subtype.ext                     --Goal : (func_to_range f x).val = y.val
  rewrite [ftr_def, h2]
  rfl
  done
```

8.1. Equinumerous Sets

Is `func_to_range f` one-to-one? It turns out that if `f` is one-to-one, then so is `func_to_range f`, and therefore `func_to_range f` can be used to prove that $U \sim \text{range } f$. Here are the proofs:

```
lemma ftr_one_one_of_one_one {U V : Type} {f : U → V}
  (h : one_to_one f) : one_to_one (func_to_range f) := by
  fix x1 : U; fix x2 : U
  assume h1 : func_to_range f x1 = func_to_range f x2
  have h2 : f x1 = f x2 :=
    calc f x1
      _ = (func_to_range f x1).val := (ftr_def f x1).symm
      _ = (func_to_range f x2).val := by rw [h1]
      _ = f x2 := ftr_def f x2
  show x1 = x2 from h x1 x2 h2
  done

theorem equinum_range {U V : Type} {f : U → V}
  (h : one_to_one f) : U ~ range f := by
  apply Exists.intro (func_to_range f)
  show one_to_one (func_to_range f) ∧ onto (func_to_range f) from
    And.intro (ftr_one_one_of_one_one h) (ftr_onto f)
  done
```

We have seen that, given a function $f : U \rightarrow V$, we can turn f into an onto function by replacing V with a subtype of V . (This is similar to an idea that is mentioned briefly in *HTPI*; see Exercise 23 in Section 5.2 of *HTPI*.) It is perhaps not surprising that we can sometimes turn f into a one-to-one function by replacing U with a subtype of U . If A has type `Set U`, then the *restriction* of f to A is a function from A to V . It has the same values as f , but only when applied to elements of A . (This idea is also mentioned in *HTPI*; see Exercise 7 in Section 5.2 of *HTPI*.) We can define the restriction of f to A as follows:

```
def func_restrict {U V : Type}
  (f : U → V) (A : Set U) (x : A) : V := f x.val

lemma fr_def {U V : Type} (f : U → V) (A : Set U) (x : A) :
  func_restrict f A x = f x.val := by rfl
```

Thus, `func_restrict f A` is a function from A to V (that is, from $\uparrow A$ to V). Is it one-to-one? The answer is: sometimes. We will say that a function is *one-to-one on A* if it satisfies the definition of one-to-one when applied to elements of A :

8.1. Equinumerous Sets

```
def one_one_on {U V : Type} (f : U → V) (A : Set U) : Prop :=
  ∀ {x1 x2 : U}, x1 ∈ A → x2 ∈ A → f x1 = f x2 → x1 = x2
```

Notice that in this definition, we have used the same double braces for the quantified variables x_1 and x_2 that were used in the definition of “subset.” This means that x_1 and x_2 are implicit arguments, and therefore if we have $h : \text{one_one_on } f A$, $ha1 : a1 \in A$, $ha2 : a2 \in A$, and $heq : f a1 = f a2$, then $h h1a h2a heq$ is a proof of $a1 = a2$. There is no need to specify that $a1$ and $a2$ are the values to be assigned to x_1 and x_2 ; Lean will figure that out for itself. (To type the double braces $\{$ and $\}$, type $\{\{$ and $\}\}$. There were cases in previous chapters where it would have been appropriate to use such implicit arguments, but we chose not to do so to avoid confusion. But by now you should be comfortable enough with Lean that you won’t be confused by this new complication.)

It is now not hard to show that if f is one-to-one on A , then $\text{func_restrict } f A$ is one-to-one:

```
lemma fr_one_one_of_one_one_on {U V : Type} {f : U → V} {A : Set U}
  (h : one_one_on f A) : one_to_one (func_restrict f A) := by
  fix x1 : A; fix x2 : A           --x1 and x2 have type ↑A
  assume h1 : func_restrict f A x1 = func_restrict f A x2
  rewrite [fr_def, fr_def] at h1      --h1 : f x1.val = f x2.val
  apply Subtype.ext                  --Goal : x1.val = x2.val
  show x1.val = x2.val from h x1.property x2.property h1
  done
```

Now we can combine our last two results: if f is one-to-one on A , then $\text{func_restrict } f A$ is one-to-one, and therefore A is equinumerous with the range of $\text{func_restrict } f A$. And what is the range of $\text{func_restrict } f A$? It is just the image of A under f :

```
lemma elt_image {U V : Type} {A : Set U} {x : U}
  (f : U → V) (h : x ∈ A) : f x ∈ image f A := by
  define                                     --Goal : ∃ x_1 ∈ A, f x_1 = f x
  apply Exists.intro x                      --Goal : x ∈ A ∧ f x = f x
  apply And.intro h
  rfl
  done

lemma fr_range {U V : Type} (f : U → V) (A : Set U) :
  range (func_restrict f A) = image f A := by
  apply Set.ext
  fix y : V
  apply Iff.intro
  · -- (→)
```

8.1. Equinumerous Sets

```

assume h1 : y ∈ range (func_restrict f A) --Goal : y ∈ image f A
define at h1
obtain (a : A) (h2 : func_restrict f A a = y) from h1
rewrite [←h2, fr_def] --Goal : f a.val ∈ image f A
show f a.val ∈ image f A from elt_image f a.property
done
· -- (←)
assume h1 : y ∈ image f A --Goal : y ∈ range (func_restrict f A)
define at h1
obtain (a : U) (h2 : a ∈ A ∧ f a = y) from h1
set aA : A := Subtype_elt h2.left
have h3 : func_restrict f A aA = f a := fr_def f A aA
rewrite [←h2.right, ←h3]
show func_restrict f A aA ∈ range (func_restrict f A) from
elt_range (func_restrict f A) aA
done
done

```

Putting it all together, we have another theorem that helps us prove that sets are equinumerous:

```

theorem equinum_image {U V : Type} {A : Set U} {f : U → V}
(h : one_one_on f A) : A ~ image f A := by
rewrite [←fr_range f A] --Goal : A ~ range (func_restrict f A)
have h1 : one_to_one (func_restrict f A) :=
fr_one_one_of_one_one_on h
show A ~ range (func_restrict f A) from equinum_range h1
done

```

We now return to the problem of showing that if A has type `Set Nat`, then it is countable. Recall that we have defined a function `num_elts_below A : Nat → Nat` that counts the number of elements of A below any natural number. Although `num_elts_below A` is not one-to-one, it turns out that it is one-to-one on A . We use this fact to show that if A has an upper bound then it is finite, and if it doesn't then it is denumerable. The details of the proof are somewhat long. We'll skip some of them here, but you can find them in the HTPI Lean package.

```

lemma neb_one_one_on (A : Set Nat) :
one_one_on (num_elts_below A) A := sorry

lemma neb_image_bdd {A : Set Nat} {m : Nat} (h : ∀ n ∈ A, n < m) :
image (num_elts_below A) A = I (num_elts_below A m) := sorry

```

```

lemma bdd_subset_nat {A : Set Nat} {m : Nat}
  (h : ∀ n ∈ A, n < m) : I (num_elts_below A m) ~ A := by
  have h2 : A ~ image (num_elts_below A) A :=
    equinum_image (neb_one_one_on A)
  rewrite [neb_image_bdd h] at h2           --h2 : A ~ I (num_elts_below A m)
  show I (num_elts_below A m) ~ A from Theorem_8_1_3_2 h2
  done

lemma neb_unbdd_onto {A : Set Nat}
  (h : ∀ (m : Nat), ∃ n ∈ A, n ≥ m) :
  onto (func_restrict (num_elts_below A) A) := sorry

lemma unbdd_subset_nat {A : Set Nat}
  (h : ∀ (m : Nat), ∃ n ∈ A, n ≥ m) :
  denum A := by
  rewrite [denum_def]
  set f : A → Nat := func_restrict (num_elts_below A) A
  have h1 : one_to_one f :=
    fr_one_one_of_one_one_on (neb_one_one_on A)
  have h2 : onto f := neb_unbdd_onto h
  have h3 : A ~ Nat := Exists.intro f (And.intro h1 h2)
  show Nat ~ A from Theorem_8_1_3_2 h3
  done

theorem set_nat_ctble (A : Set Nat) : ctble A := by
  define           --Goal : finite A ∨ denum A
  by_cases h1 : ∃ (m : Nat), ∀ n ∈ A, n < m
  . -- Case 1. h1 : ∃ (m : Nat), ∀ n ∈ A, n < m
    apply Or.inl --Goal : finite A
    obtain (m : Nat) (h2 : ∀ n ∈ A, n < m) from h1
    define
      apply Exists.intro (num_elts_below A m)
      show I (num_elts_below A m) ~ A from bdd_subset_nat h2
      done
  . -- Case 2. h1 : ¬∃ (m : Nat), ∀ n ∈ A, n < m
    apply Or.inr --Goal : denum A
    push_neg at h1
    --This tactic converts h1 to ∀ (m : Nat), ∃ n ∈ A, m ≤ n
    show denum A from unbdd_subset_nat h1
    done
  done

```

8.1. Equinumerous Sets

As a consequence of our last theorem, we will get another characterization of countability: a type is countable if and only if it is equinumerous with some set of natural numbers. A finite type is equinumerous with the set I_n , for some natural number n . But what set of natural numbers is a denumerable type equinumerous with? The obvious choice is the set of all natural numbers. We might call it the *universal set* for the type `Nat`. It will be convenient to have notation for the universal set of any type. It is not hard to show that any type is equinumerous with its universal set. We leave one lemma for the proof as an exercise for you:

```
def Univ (U : Type) : Set U := {x : U | True}

lemma elt_univ {U : Type} (x : U) : x ∈ Univ U := by
  define   --Goal : True
  trivial
  done

lemma onto_iff_range_eq_univ {U V : Type} (f : U → V) :
  onto f ↔ range f = Univ V := sorry

lemma univ_equinum_type (U : Type) : Univ U ~ U := by
  set f : U → U := id
  have h1 : one_to_one f := id_one_one U
  have h2 : onto f := id_onto U
  rewrite [onto_iff_range_eq_univ] at h2  --h2 : range f = Univ U
  have h3 : U ~ range f := equinum_range h1
  rewrite [h2] at h3
  show Univ U ~ U from Theorem_8_1_3_2 h3
  done
```

With this preparation, we can prove our next characterization of countability, leaving the proof of a lemma as an exercise for you:

```
lemma ctable_of_ctble_equinum {U V : Type}
  (h1 : U ~ V) (h2 : ctable U) : ctable V := sorry

theorem ctable_iff_set_nat_equinum (U : Type) :
  ctable U ↔ ∃ (J : Set Nat), J ~ U := by
  apply Iff.intro
  · -- (→)
    assume h1 : ctable U
    define at h1  --h1 : finite U ∨ denum U
    by_cases on h1
    · -- Case 1. h1 : finite U
```

```

define at h1  --h1 : ∃ (n : Nat), I n ~ U
obtain (n : Nat) (h2 : I n ~ U) from h1
show ∃ (J : Set Nat), J ~ U from Exists.intro (I n) h2
done
• -- Case 2. h1 : denum U
  rewrite [denum_def] at h1  --h1 : Nat ~ U
  have h2 : Univ Nat ~ Nat := univ_equinum_type Nat
  apply Exists.intro (Univ Nat)
  show Univ Nat ~ U from Theorem_8_1_3_3 h2 h1
  done
done
done
• -- (→)
  assume h1 : ∃ (J : Set Nat), J ~ U
  obtain (J : Set Nat) (h2 : J ~ U) from h1
  have h3 : ctable J := set_nat_ctble J
  show ctable U from ctable_of_ctble_equinum h2 h3
  done
done

```

We have seen that if we have $f : U \rightarrow V$, then it can be useful to replace either U or V with a subtype. We accomplished this by defining two functions, `func_to_range` and `func_restrict`. It is also sometimes useful to go in the other direction. To do this, we will define functions `func_to_type` and `func_extend` that are, in a sense, the reverses of `func_to_range` and `func_restrict`. If we have $f : U \rightarrow B$, where B has type `Set V`, then `func_to_type f` is a function from U to V ; it is the same as f , but with the values of the function coerced from B to V . If we have $f : A \rightarrow V$ and $v : V$, where A has type `Set U`, then `func_extend f v` is also a function from U to V ; when applied to an element of A , it has the value specified by f , and when applied to anything else it has the value v . Here are the definitions:

```

def func_to_type {U V : Type} {B : Set V}
  (f : U → B) (x : U) : V := (f x).val

lemma ftt_def {U V : Type} {B : Set V} (f : U → B) (x : U) :
  func_to_type f x = (f x).val := by rfl

noncomputable def func_extend {U V : Type} {A : Set U}
  (f : A → V) (v : V) (u : U) : V :=
  if test : u ∈ A then f (Subtype.elt test) else v

lemma fe_elt {U V : Type} {A : Set U} (f : A → V) (v : V) (a : A) :
  func_extend f v a.val = f a := dif_pos a.property

```

8.1. Equinumerous Sets

Notice that in the definition of `func_extend`, we gave an identifier to the test in the `if` clause, so that we could refer to that test in the `then` clause. As a result, the `if-then-else` expression in the definition is what is called a *dependent if-then-else*. The proof of the lemma `fe_elt` uses the theorem `dif_pos`, which is just like `if_pos`, but for dependent if-then-else expressions. (Of course, `dif_neg` is the dependent version of `if_neg`.)

The sense in which `func_to_type` and `func_extend` reverse `func_to_range` and `func_restrict` is given by the following examples. We leave the proof of the second as an exercise for you:

```
example (U V : Type) (f : U → V) :
  func_to_type (func_to_range f) = f := by rfl

example (U V : Type) (A : Set U) (f : A → V) (v : V) :
  func_restrict (func_extend f v) A = f := sorry
```

How do `func_to_type` and `func_extend` interact with the properties `one_to_one` and `onto`? The answers are given by the following lemmas; you can find the straightforward proofs in the HTPI Lean package.

```
lemma ftt_range_of_onto {U V : Type} {B : Set V} {f : U → B}
  (h : onto f) : range (func_to_type f) = B := sorry

lemma ftt_one_one_of_one_one {U V : Type} {B : Set V} {f : U → B}
  (h : one_to_one f) : one_to_one (func_to_type f) := sorry

lemma fe_image {U V : Type} {A : Set U}
  (f : A → V) (v : V) : image (func_extend f v) A = range f := sorry

lemma fe_one_one_on_of_one_one {U V : Type} {A : Set U} {f : A → V}
  (h : one_to_one f) (v : V) : one_one_on (func_extend f v) A := sorry
```

Just as `func_to_type` and `func_extend` can be thought of as reversing `func_to_range` and `func_restrict`, the following theorem can be thought of as reversing the theorem `equinum_image`:

```
theorem type_to_type_of_equinum {U V : Type} {A : Set U} {B : Set V}
  (h : A ~ B) (v : V) :
  ∃ (f : U → V), one_one_on f A ∧ image f A = B := by
  obtain (g : A → B) (h1 : one_to_one g ∧ onto g) from h
  set gtt : A → V := func_to_type g
  set f : U → V := func_extend gtt v
  apply Exists.intro f
```

```

apply And.intro
• -- Proof of one_to_one_on f A
  have h2 : one_to_one gtt := ftt_one_one_of_one_one h1.left
  show one_one_on f A from fe_one_one_on_of_one_one h2 v
  done
• -- Proof of image f A = B
  have h2 : range gtt = B := ftt_range_of_onto h1.right
  have h3 : image f A = range gtt := fe_image gtt v
  rewrite [h3, h2]
  rfl
  done
done

```

We are finally ready to turn to Theorem 8.1.5 in *HTPI*. The theorem says that for any set A , the following statements are equivalent:

1. A is countable.
2. Either $A = \emptyset$ or there is a function $f : \mathbb{Z}^+ \rightarrow A$ that is onto.
3. There is a function $f : A \rightarrow \mathbb{Z}^+$ that is one-to-one.

We will find it convenient to phrase these statements a little differently in Lean. Our plan is to prove that if A has type `Set U` then the following statements are equivalent:

1. `ctble A`
2. `empty A v ∃ (f : Nat → U), A ⊆ range f`
3. `∃ (f : U → Nat), one_one_on f A`

As in *HTPI*, we will do this by proving $1 \rightarrow 2 \rightarrow 3 \rightarrow 1$. Here is the proof of $1 \rightarrow 2$.

```

theorem Theorem_8_1_5_1_to_2 {U : Type} {A : Set U} (h : ctle A) :
  empty A v ∃ (f : Nat → U), A ⊆ range f := by
  or_right with h1
  obtain (a : U) (h2 : a ∈ A) from h1
  rewrite [ctble_iff_set_nat_equinum] at h --h : ∃ (J : Set Nat), J ~ A
  obtain (J : Set Nat) (h3 : J ~ A) from h
  obtain (f : Nat → U) (h4 : one_one_on f J ∧ image f J = A) from
    type_to_type_of_equinum h3 a
  apply Exists.intro f --Goal : A ⊆ range f
  fix y : U
  assume h5 : y ∈ A
  rewrite [←h4.right] at h5
  define at h5
  obtain (n : Nat) (h6 : n ∈ J ∧ f n = y) from h5

```

```

rewrite [←h6.right]
show f n ∈ range f from elt_range f n
done

```

For the proof of 2 → 3, we need to consider two cases. If A is empty, then any function $f : U \rightarrow \text{Nat}$ will do to prove statement 3; we use the constant function that always takes the value 0. The easiest way to prove that this function is one-to-one on A is to use a lemma that says if A is empty, then any statement of the form $x \in A \rightarrow P$ is true.

The more interesting case is when we have a function $g : \text{Nat} \rightarrow U$ with $A \subseteq \text{range } g$. This means that for each $x \in A$, there is at least one natural number n such that $g n = x$. As in *HTPI*, we first define a function $F : A \rightarrow \text{Nat}$ that picks out the *smallest* such n; we could call it the *smallest preimage of x*. We then use `func_extend` to get the required function from U to `Nat`. The easiest way to define F is to first define its graph, which we call `smallest_preimage_graph g A`.

```

def constant_func (U : Type) {V : Type} (v : V) (x : U) : V := v

lemma elt_empty_implies {U : Type} {A : Set U} {x : U} {P : Prop}
  (h : empty A) : x ∈ A → P := by
  assume h1 : x ∈ A
  contradict h
  show ∃ (x : U), x ∈ A from Exists.intro x h1
  done

lemma one_one_on_empty {U : Type} {A : Set U}
  (f : U → Nat) (h : empty A) : one_one_on f A := by
  fix x1 : U; fix x2 : U
  show x1 ∈ A → x2 ∈ A → f x1 = f x2 → x1 = x2 from
    elt_empty_implies h
  done

def smallest_preimage_graph {U : Type}
  (g : Nat → U) (A : Set U) : Set (A × Nat) :=
  {(x, n) : A × Nat | g n = x.val ∧ ∀ (m : Nat), g m = x.val → n ≤ m}

lemma spg_is_func_graph {U : Type} {g : Nat → U} {A : Set U}
  (h : A ⊆ range g) : is_func_graph (smallest_preimage_graph g A) := by
  define
  fix x : A
  exists_unique
  • -- Existence

```

```

set W : Set Nat := {n : Nat | g n = x.val}
have h1 : ∃ (n : Nat), n ∈ W := h x.property
show ∃ (y : Nat), (x, y) ∈ smallest_preimage_graph g A from
  well_ord_princ W h1
done
• -- Uniqueness
fix n1 : Nat; fix n2 : Nat
assume h1 : (x, n1) ∈ smallest_preimage_graph g A
assume h2 : (x, n2) ∈ smallest_preimage_graph g A
define at h1; define at h2
have h3 : n1 ≤ n2 := h1.right n2 h2.left
have h4 : n2 ≤ n1 := h2.right n1 h1.left
linarith
done
done

lemma spg_one_one {U : Type} {g : Nat → U} {A : Set U} {f : A → Nat}
  (h : graph f = smallest_preimage_graph g A) : one_to_one f := by
fix a1 : A; fix a2 : A
assume h1 : f a1 = f a2
set y : Nat := f a2           --h1 : f a1 = y
have h2 : f a2 = y := by rfl
rewrite [←graph_def, h] at h1 --h1 : (a1, y) ∈ smallest_preimage_graph g A
rewrite [←graph_def, h] at h2 --h2 : (a2, y) ∈ smallest_preimage_graph g A
define at h1           --h1 : g y = a1.val ∧ ...
define at h2           --h2 : g y = a2.val ∧ ...
apply Subtype.ext        --Goal : a1.val = a2.val
rewrite [←h1.left, ←h2.left]
rfl
done

theorem Theorem_8_1_5_2_to_3 {U : Type} {A : Set U}
  (h : empty A ∨ ∃ (f : Nat → U), A ⊆ range f) :
  ∃ (f : U → Nat), one_one_on f A := by
by_cases on h
• -- Case 1. h : empty A
  set f : U → Nat := constant_func U 0
  apply Exists.intro f
  show one_one_on f A from one_one_on_empty f h
  done
• -- Case 2. h : ∃ (f : Nat → U), A ⊆ range f
  obtain (g : Nat → U) (h1 : A ⊆ range g) from h

```

8.1. Equinumerous Sets

```

have h2 : is_func_graph (smallest_preimage_graph g A) :=
  spg_is_func_graph h1
rewrite [←func_from_graph] at h2
obtain (F : A → Nat)
  (h3 : graph F = smallest_preimage_graph g A) from h2
have h4 : one_to_one F := spg_one_one h3
set f : U → Nat := func_extend F 0
apply Exists.intro f
show one_one_on f A from fe_one_one_on_of_one_one h4 0
done
done

```

Finally, the proof of 3 → 1 is straightforward, using the theorem equinum_image.

```

theorem Theorem_8_1_5_3_to_1 {U : Type} {A : Set U}
  (h1 : ∃ (f : U → Nat), one_one_on f A) :
  ctable A := by
obtain (f : U → Nat) (h2 : one_one_on f A) from h1
have h3 : A ~ image f A := equinum_image h2
rewrite [ctble_iff_set_nat_equinum]
show ∃ (J : Set Nat), J ~ A from
  Exists.intro (image f A) (Theorem_8_1_3_2 h3)
done

```

We now know that statements 1–3 are equivalent, which means that statements 2 and 3 can be thought of as alternative ways to think about countability:

```

theorem Theorem_8_1_5_2 {U : Type} (A : Set U) :
  ctable A ↔ empty A ∨ ∃ (f : Nat → U), A ⊆ range f := by
apply Iff.intro Theorem_8_1_5_1_to_2
assume h1 : empty A ∨ ∃ (f : Nat → U), A ⊆ range f
have h2 : ∃ (f : U → Nat), one_one_on f A := Theorem_8_1_5_2_to_3 h1
show ctable A from Theorem_8_1_5_3_to_1 h2
done

theorem Theorem_8_1_5_3 {U : Type} (A : Set U) :
  ctable A ↔ ∃ (f : U → Nat), one_one_on f A := sorry

```

We end this section with a proof of Theorem 8.1.6 in *HTPI*, which says that the set of rational numbers is denumerable. Our strategy is to define a one-to-one function from `Rat` (the type of rational numbers) to `Nat`. We will need to know a little bit about the way rational numbers

are represented in Lean. If q has type `Rat`, then when q is written in lowest terms, $q.num$ is the numerator, which is an integer, and $q.den$ is the denominator, which is a nonzero natural number. The theorem `Rat.ext` says that if two rational numbers have the same numerator and denominator, then they are equal. And we will also use the theorem `Prod.mk.inj`, which says that if two ordered pairs are equal, then their first coordinates are equal, as are their second coordinates.

```
def fqn (q : Rat) : Nat := fnnn (fzn q.num, q.den)

lemma fqn_def (q : Rat) : fqn q = fnnn (fzn q.num, q.den) := by rfl

lemma fqn_one_one : one_to_one fqn := by
  define
  fix q1 : Rat; fix q2 : Rat
  assume h1 : fqn q1 = fqn q2
  rewrite [fqn_def, fqn_def] at h1
  --h1 : fnnn (fzn q1.num, q1.den) = fnnn (fzn q2.num, q2.den)
  have h2 : (fzn q1.num, q1.den) = (fzn q2.num, q2.den) :=
    fnnn_one_one _ _ h1
  have h3 : fzn q1.num = fzn q2.num ∧ q1.den = q2.den :=
    Prod.mk.inj h2
  have h4 : q1.num = q2.num := fnn_one_one _ _ h3.left
  show q1 = q2 from Rat.ext h4 h3.right
  done

lemma range_fqn_unbdd :
  ∀ (m : Nat), ∃ n ∈ range fqn, n ≥ m := by
  fix m : Nat
  set n : Nat := fqn ↑m
  apply Exists.intro n
  apply And.intro
  · -- Proof that n ∈ range fqn
    define
    apply Exists.intro ↑m
    rfl
    done
  · -- Proof that n ≥ m
    show n ≥ m from
      calc n
        _ = tri (2 * m + 1) + 2 * m := by rfl
        _ ≥ m := by linarith
    done
  done
```

```
theorem Theorem_8_1_6 : denum Rat := by
  set I : Set Nat := range fqn
  have h1 : Nat ~ I := unbdd_subset_nat range_fqn_unbdd
  have h2 : Rat ~ I := equinum_range fqn_one_one
  have h3 : I ~ Rat := Theorem_8_1_3_2 h2
  show denum Rat from Theorem_8_1_3_3 h1 h3
  done
```

Exercises

1. --Hint: Use Exercise_6_1_16a2 from the exercises of Section 6.1.

```
lemma fnz_odd (k : Nat) : fnz (2 * k + 1) = -↑(k + 1) := sorry
```

2. **lemma** fnz_fnz : fnz ∘ fnz = id := **sorry**

3. **lemma** tri_step (k : Nat) : tri (k + 1) = tri k + k + 1 := **sorry**

4. **lemma** tri_incr {j k : Nat} (h1 : j ≤ k) : tri j ≤ tri k := **sorry**

5. **example** {U V : Type} (f : U → V) : range f = Ran (graph f) := **sorry**

6. **lemma** onto_iff_range_eq_univ {U V : Type} (f : U → V) :
 onto f ↔ range f = Univ V := **sorry**

7. Notice that ctable_of_ctble_equinum was used in the proof of ctable_iff_set_nat_equinum. Therefore, to avoid circularity, you must not use ctable_iff_set_nat_equinum in your solution to this exercise.

```
lemma ctable_of_ctble_equinum {U V : Type}
  (h1 : U ~ V) (h2 : ctable U) : ctable V := sorry
```

8. **theorem** Exercise_8_1_1_b : denum {n : Int | even n} := **sorry**

9. **theorem** equinum_iff_inverse_pair (U V : Type) :
 U ~ V ↔ ∃ (f : U → V) (g : V → U), f ∘ g = id ∧ g ∘ f = id := **sorry**

10. Notice that if f is a function from U to V , then for every X of type $\text{Set } U$, $\text{image } f X$ has type $\text{Set } V$. Therefore $\text{image } f$ is a function from $\text{Set } U$ to $\text{Set } V$.

```
lemma image_comp_id {U V : Type} {f : U → V} {g : V → U}
  (h : g ∘ f = id) : (image g) ∘ (image f) = id := sorry
```

11. `theorem Exercise_8_1_5_1 {U V : Type}
(h : U ~ V) : Set U ~ Set V := sorry`

If A has type $\text{Set } U$ and X has type $\text{Set } A$ (that is, $\text{Set } \uparrow A$), then every element of X has type $\uparrow A$, which means that its value is an element of A . Thus, if we take the values of all the elements of X , we will get a set of type $\text{Set } U$ that is a subset of A . We will call this the *value image* of X . We can define a function that computes the value image of any $X : \text{Set } A$:

```
def val_image {U : Type} (A : Set U) (X : Set A) : Set U :=  
{y : U | ∃ x ∈ X, x.val = y}
```

The next three exercises ask you to prove properties of this function.

12. `lemma subset_of_val_image_eq {U : Type} {A : Set U} {X1 X2 : Set A}
(h : val_image A X1 = val_image A X2) : X1 ⊆ X2 := sorry`

13. `lemma val_image_one_one {U : Type} (A : Set U) :
one_to_one (val_image A) := sorry`

14. `lemma range_val_image {U : Type} (A : Set U) :
range (val_image A) = ℙ A := sorry`

15. `lemma Set_equinum_powerset {U : Type} (A : Set U) :
Set A ~ ℙ A := sorry`

16. --Hint: Use Exercise_8_1_5_1 and Set_equinum_powerset.
`theorem Exercise_8_1_5_2 {U V : Type} {A : Set U} {B : Set V}
(h : A ~ B) : ℙ A ~ ℙ B := sorry`

17. `example (U V : Type) (A : Set U) (f : A → V) (v : V) :
func_restrict (func_extend f v) A = f := sorry`

18. We proved the implications in Theorem 8.1.5 for sets, but we could prove similar theorems for types. Here is a version of Theorem_8_1_5_3 for types.

```
theorem Theorem_8_1_5_3_type {U : Type} :  
ctble U ↔ ∃ (f : U → Nat), one_to_one f := sorry
```

19. `theorem ctable_set_of_ctable_type {U : Type}
(h : ctable U) (A : Set U) : ctable A := sorry`

20. `theorem Exercise_8_1_17 {U : Type} {A B : Set U}
(h1 : B ⊆ A) (h2 : ctable A) : ctable B := sorry`

8.1½. Debts Paid

It is time to fulfill promises we made in two earlier chapters.

In Section 6.2, we promised to define a proposition `numElts A n` to express the idea that the set `A` has `n` elements. It should now be clear how to define this proposition:

```
def numElts {U : Type} (A : Set U) (n : Nat) : Prop := I n ~ A

lemma numElts_def {U : Type} (A : Set U) (n : Nat) :
  numElts A n ↔ I n ~ A := by rfl
```

It is sometimes convenient to phrase the definition of `finite` in terms of `numElts`, so we state that version of the definition as a lemma.

```
lemma finite_def {U : Type} (A : Set U) :
  finite A ↔ ∃ (n : Nat), numElts A n := by rfl
```

We also owe you the proofs of several theorems about `numElts`. We'll skip the details of some of these proofs, but for those that are not left as exercises, you can find all the details in the HTPi Lean package. We begin with the fact that a set has zero elements if and only if it is empty. If `A` has type `Set U` and `A` is empty, then any function from `U` to `Nat` can be used to prove `A ~ I 0`. In the proof below, we use a constant function.

```
lemma not_empty_iff_exists_elt {U : Type} {A : Set U} :
  ¬empty A ↔ ∃ (x : U), x ∈ A := by
  define : empty A
  double_neg
  rfl
  done

lemma image_empty {U : Type} {A : Set U}
  (f : U → Nat) (h : empty A) : image f A = I 0 := sorry

theorem zero_elts_iff_empty {U : Type} (A : Set U) :
  numElts A 0 ↔ empty A := by
  apply Iff.intro
  · -- (→)
    assume h1 : numElts A 0
    define at h1
    obtain (f : I 0 → A) (h2 : one_to_one f ∧ onto f) from h1
    by_contra h3
```

```

rewrite [not_empty_iff_exists_elt] at h3
obtain (x : U) (h4 : x ∈ A) from h3
set xA : A := Subtype_elt h4
obtain (n : I 0) (h5 : f n = xA) from h2.right xA
have h6 : n.val < 0 := n.property
linarith
done
• -- (→)
assume h1 : empty A
rewrite [numElts_def]
set f : U → Nat := constant_func U 0
have h2 : one_one_on f A := one_one_on_empty f h1
have h3 : image f A = I 0 := image_empty f h1
have h4 : A ~ image f A := equinum_image h2
rewrite [h3] at h4
show I 0 ~ A from Theorem_8_1_3_2 h4
done
done

```

Next, we prove that if a set has a positive number of elements then it is not empty. The proof is straightforward.

```

theorem nonempty_of_pos_numElts {U : Type} {A : Set U} {n : Nat}
  (h1 : numElts A n) (h2 : n > 0) : ∃ (x : U), x ∈ A := by
  define at h1
  obtain (f : I n → A) (h3 : one_to_one f ∧ onto f) from h1
  have h4 : 0 ∈ I n := h2
  set x : A := f (Subtype_elt h4)
  show ∃ (x : U), x ∈ A from Exists.intro x.val x.property
  done

```

Our next theorem is `remove_one_numElts`, which says that if a set has $n + 1$ elements, and we remove one element, then the resulting set has n elements. We begin by proving that for any $k < n + 1$, if we remove k from $I(n + 1)$ then the resulting set is equinumerous with $I n$. To do this, we define a function that matches up $I n$ with $I(n + 1) \setminus \{k\}$.

```
def incr_from (k n : Nat) : Nat := if n < k then n else n + 1
```

The function `incr_from k` increments natural numbers from k on, while leaving numbers less than k fixed. Our strategy now is to prove that `incr_from k` is one-to-one on $I n$, and the image of $I n$ under `incr_from k` is $I(n + 1) \setminus \{k\}$. The proof is a bit long, so we skip some

of the details. Notice that Lean gets confused when coercing $I(n + 1) \setminus \{k\}$ to a subtype unless we specify that we want $\uparrow(I(n + 1) \setminus \{k\})$ rather than $\uparrow(I(n + 1)) \setminus \uparrow\{k\}$.

```
lemma incr_from_one_one (k : Nat) :
  one_to_one (incr_from k) := sorry

lemma incr_from_image {k n : Nat} (h : k < n + 1) :
  image (incr_from k) (I n) = I (n + 1) \ \{k\} := sorry

lemma one_one_on_of_one_one {U V : Type} {f : U → V}
  (h : one_to_one f) (A : Set U) : one_one_on f A := by
  define
  fix x1 : U; fix x2 : U
  assume h1 : x1 ∈ A
  assume h2 : x2 ∈ A
  show f x1 = f x2 → x1 = x2 from h x1 x2
  done

lemma I_equinum_I_remove_one {k n : Nat}
  (h : k < n + 1) : I n ~ \uparrow(I (n + 1) \ \{k\}) := by
  rewrite [←incr_from_image h]
  show I n ~ image (incr_from k) (I n) from
    equinum_image (one_one_on_of_one_one (incr_from_one_one k) (I n))
  done
```

Using one more lemma, whose proof we leave as an exercise for you, we can prove `remove_one_numElts`.

```
lemma remove_one_equinum
  {U V : Type} {A : Set U} {B : Set V} {a : U} {b : V} {f : U → V}
  (h1 : one_one_on f A) (h2 : image f A = B)
  (h3 : a ∈ A) (h4 : f a = b) : \uparrow(A \ \{a\}) ~ \uparrow(B \ \{b\}) := sorry

theorem remove_one_numElts {U : Type} {A : Set U} {n : Nat} {a : U}
  (h1 : numElts A (n + 1)) (h2 : a ∈ A) : numElts (A \ \{a\}) n := by
  rewrite [numElts_def] at h1; rewrite [numElts_def]
  --h1 : I (n + 1) ~ A; Goal : I n ~ \uparrow(A \ \{a\})
  obtain (f : Nat → U) (h3 : one_one_on f (I (n + 1)) ∧
    image f (I (n + 1)) = A) from type_to_type_of_equinum h1 a
  rewrite [←h3.right] at h2
  obtain (k : Nat) (h4 : k ∈ I (n + 1) ∧ f k = a) from h2
  have h5 : \uparrow(I (n + 1) \ \{k\}) ~ \uparrow(A \ \{a\}) :=
```

```

remove_one_equinum h3.left h3.right h4.left h4.right
have h6 : k < n + 1 := h4.left
have h7 : I n ~ ↑(I (n + 1) \ {k}) := I_equinum_I_remove_one h6
show I n ~ ↑(A \ {a}) from Theorem_8_1_3_3 h7 h6
done

```

Finally, we prove that a set has one element if and only if it is a singleton set, leaving the proof of one lemma as an exercise for you.

```

lemma singleton_of_diff_empty {U : Type} {A : Set U} {a : U}
  (h1 : a ∈ A) (h2 : empty (A \ {a})) : A = {a} := sorry

lemma one_one_on_I_1 {U : Type} (f : Nat → U) : one_one_on f (I 1) := by
  fix x1 : Nat; fix x2 : Nat
  assume h1 : x1 ∈ I 1
  assume h2 : x2 ∈ I 1
  assume h3 : f x1 = f x2
  define at h1; define at h2 --h1 : x1 < 1; h2 : x2 < 1
  linarith
  done

lemma image_I_1 {U : Type} (f : Nat → U) : image f (I 1) = {f 0} := by
  apply Set.ext
  fix y
  apply Iff.intro
  · -- (→)
    assume h1 : y ∈ image f (I 1)
    define at h1; define
    obtain (x : Nat) (h2 : x ∈ I 1 ∧ f x = y) from h1
    have h3 : x < 1 := h2.left
    have h4 : x = 0 := by linarith
    rewrite [←h2.right, h4]
    rfl
    done
  · -- (←)
    assume h1 : y ∈ {f 0}
    define at h1; define
    apply Exists.intro 0
    apply And.intro _ h1.symm
    define
    linarith
    done

```

```

done

lemma singleton_one_elt {U : Type} (u : U) : numElts {u} 1 := by
  rewrite [numElts_def] --Goal : I 1 ~ {u}
  set f : Nat → U := constant_func Nat u
  have h1 : one_one_on f (I 1) := one_one_on_I_1 f
  have h2 : image f (I 1) = {f 0} := image_I_1 f
  have h3 : f 0 = u := by rfl
  rewrite [←h3, ←h2]
  show I 1 ~ image f (I 1) from equinum_image h1
done

theorem one_elt_iff_singleton {U : Type} (A : Set U) :
  numElts A 1 ↔ ∃ (x : U), A = {x} := by
  apply Iff.intro
  · -- (→)
    assume h1 : numElts A 1 --Goal : ∃ (x : U), A = {x}
    have h2 : 1 > 0 := by linarith
    obtain (x : U) (h3 : x ∈ A) from nonempty_of_pos_numElts h1 h2
    have h4 : numElts (A \ {x}) 0 := remove_one_numElts h1 h3
    rewrite [zero_elts_iff_empty] at h4
    show ∃ (x : U), A = {x} from
      Exists.intro x (singleton_of_diff_empty h3 h4)
    done
  · -- (←)
    assume h1 : ∃ (x : U), A = {x}
    obtain (x : U) (h2 : A = {x}) from h1
    rewrite [h2]
    show numElts {x} 1 from singleton_one_elt x
    done
done

```

We have now proven all of the theorems about `numElts` whose proofs were promised in Section 6.2. However, there is still one important issue that we have not addressed. Could there be a set A such that, say, $\text{numElts } A \ 5$ and $\text{numElts } A \ 6$ are both true? Surely the answer is no—a set can't have five elements and also have six elements! But it requires proof. We ask you to prove the following theorem in the exercises.

```

theorem Exercise_8_1_6b {U : Type} {A : Set U} {m n : Nat}
  (h1 : numElts A m) (h2 : numElts A n) : m = n := sorry

```

Next, we turn to our promise, at the end of Section 7.4, to prove Theorem 7.4.4 of *HTPI*,

which says that the totient function φ is multiplicative.

To define the totient function in Lean, in Chapter 7 we defined `phi m` to be `num_rp_below m m`, where `num_rp_below m k` is the number of natural numbers less than k that are relatively prime to m . But in this chapter we have developed new methods for counting things. Our first task is to show that these new methods agree with the method used in Chapter 7.

We have already remarked that the definition of `num_elts_below` in this chapter bears some resemblance to the definition of `num_rp_below` in Chapter 7. It should not be surprising, therefore, that these two counting methods give results that agree.

```
def set_rp_below (m : Nat) : Set Nat := {n : Nat | rel_prime m n ∧ n < m}

lemma set_rp_below_def (a m : Nat) :
  a ∈ set_rp_below m ↔ rel_prime m a ∧ a < m := by rfl

lemma neb_nrbp (m : Nat) : ∀ {k : Nat}, k ≤ m →
  num_elts_below (set_rp_below m) k = num_rp_below m k := sorry

lemma neb_phi (m : Nat) :
  num_elts_below (set_rp_below m) m = phi m := by
  rewrite [phi_def]
  have h1 : m ≤ m := by linarith
  show num_elts_below (set_rp_below m) m = num_rp_below m m from
    neb_nrbp m h1
  done

lemma phi_is_numElts (m : Nat) :
  numElts (set_rp_below m) (phi m) := by
  rewrite [numElts_def, ←neb_phi m]
  --Goal : I (num_elts_below (set_rp_below m) m) ~ set_rp_below m
  have h1 : ∀ n ∈ set_rp_below m, n < m := by
    fix n : Nat
    assume h2 : n ∈ set_rp_below m
    define at h2
    show n < m from h2.right
    done
  show I (num_elts_below (set_rp_below m) m) ~ set_rp_below m from
    bdd_subset_nat h1
  done
```

According to the last lemma, we can now think of `phi m` as the number of elements of the set `set_rp_below m`.

We will need one more number-theoretic fact: Lemma 7.4.7 from *HTPI*. We follow the strategy of the proof in *HTPI*, separating out one calculation as an auxiliary lemma before giving the main proof.

```

lemma Lemma_7_4_7_aux {m n : Nat} {s t : Int}
  (h : s * m + t * n = 1) (a b : Nat) :
  t * n * a + s * m * b ≡ a (MOD m) := by
  define
  apply Exists.intro (s * (b - a))
  show t * n * a + s * m * b - a = m * (s * (b - a)) from
  calc t * n * a + s * m * b - a
    _ = (t * n - 1) * a + s * m * b := by ring
    _ = (t * n - (s * m + t * n)) * a + s * m * b := by rw [h]
    _ = m * (s * (b - a)) := by ring
  done

lemma Lemma_7_4_7 {m n : Nat} [NeZero m] [NeZero n]
  (h1 : rel_prime m n) (a b : Nat) :
  ∃ (r : Nat), r < m * n ∧ r ≡ a (MOD m) ∧ r ≡ b (MOD n) := by
  set s : Int := gcd_c1 m n
  set t : Int := gcd_c2 m n
  have h4 : s * m + t * n = gcd m n := gcd_lin_comb n m
  define at h1                      --h1 : gcd m n = 1
  rewrite [h1, Nat.cast_one] at h4   --h4 : s * m + t * n = 1
  set x : Int := t * n * a + s * m * b
  have h5 : x ≡ a (MOD m) := Lemma_7_4_7_aux h4 a b
  rewrite [add_comm] at h4           --h4 : t * n + s * m = 1
  have h6 : s * m * b + t * n * a ≡ b (MOD n) :=
    Lemma_7_4_7_aux h4 b a
  have h7 : s * m * b + t * n * a = x := by ring
  rewrite [h7] at h6                --h6 : x ≡ b (MOD n)
  have h8 : m * n ≠ 0 := mul_ne_zero (NeZero.ne m) (NeZero.ne n)
  rewrite [←neZero_iff] at h8       --h8 : NeZero (m * n)
  have h9 : 0 ≤ x % ↑(m * n) ∧ x % ↑(m * n) < ↑(m * n) ∧
    x ≡ x % ↑(m * n) (MOD m * n) := mod_cmpl_res (m * n) x
  have h10 : x % ↑(m * n) < ↑(m * n) ∧
    x ≡ x % ↑(m * n) (MOD m * n) := h9.right
  set r : Nat := Int.toNat (x % ↑(m * n))
  have h11 : x % ↑(m * n) = ↑r := (Int.toNat_of_nonneg h9.left).symm
  rewrite [h11, Nat.cast_lt] at h10 --h10 : r < m * n ∧ x ≡ r (MOD m * n)
  apply Exists.intro r
  apply And.intro h10.left
  have h12 : r ≡ x (MOD (m * n)) := congr_symm h10.right

```

```

rewrite [Lemma_7_4_5 _ _ h1] at h12 --h12 : r ≡ x (MOD m) ∧ r ≡ x (MOD n)
apply And.intro
· -- Proof that r ≡ a (MOD m)
  show r ≡ a (MOD m) from congr_trans h12.left h5
  done
· -- Proof that r ≡ b (MOD n)
  show r ≡ b (MOD n) from congr_trans h12.right h6
  done
done

```

The next fact we need is part 1 of Theorem 8.1.2 in *HTPI*, which says that if A , B , C , and D are sets such that $A \sim B$ and $C \sim D$, then $A \times C \sim B \times D$. It is straightforward to translate the *HTPI* proof into a Lean proof about Cartesian products of equinumerous types.

```

def func_prod {U V W X : Type} (f : U → V) (g : W → X)
  (p : U × W) : V × X := (f p.1, g p.2)

lemma func_prod_def {U V W X : Type}
  (f : U → V) (g : W → X) (u : U) (w : W) :
  func_prod f g (u, w) = (f u, g w) := by rfl

theorem Theorem_8_1_2_1_type {U V W X : Type}
  (h1 : U ~ V) (h2 : W ~ X) : (U × W) ~ (V × X) := by
  obtain (f : U → V) (h3 : one_to_one f ∧ onto f) from h1
  obtain (g : W → X) (h4 : one_to_one g ∧ onto g) from h2
  apply Exists.intro (func_prod f g)
  apply And.intro
  · -- Proof of one_to_one (func_prod f g)
    fix (u1, w1) : U × W; fix (u2, w2) : U × W
    assume h5 : func_prod f g (u1, w1) = func_prod f g (u2, w2)
    rewrite [func_prod_def, func_prod_def] at h5
    have h6 : f u1 = f u2 ∧ g w1 = g w2 := Prod.mk.inj h5
    have h7 : u1 = u2 := h3.left u1 u2 h6.left
    have h8 : w1 = w2 := h4.left w1 w2 h6.right
    rewrite [h7, h8]
    rfl
    done
  · -- Proof of onto (func_prod f g)
    fix (v, x) : V × X
    obtain (u : U) (h5 : f u = v) from h3.right v
    obtain (w : W) (h6 : g w = x) from h4.right x
    apply Exists.intro (u, w)

```

```
rewrite [func_prod_def, h5, h6]
rfl
done
```

Using coercions to subtypes, we can also apply this theorem to sets. If A , B , C , and D are sets and we have $A \sim B$ and $C \sim D$, then [Theorem_8_1_2_1_type](#) implies that $\uparrow A \times \uparrow C \sim \uparrow B \times \uparrow D$. Unfortunately, Cartesian products of subtypes are somewhat inconvenient to work with. It will turn out to be easier to work with subtypes of Cartesian products. To make this possible, we define a Cartesian product operation on sets:

```
def set_prod {U V : Type} (A : Set U) (B : Set V) : Set (U × V) :=
{(a, b) : U × V | a ∈ A ∧ b ∈ B}

notation:75 A:75 " ×s " B:75 => set_prod A B

lemma set_prod_def {U V : Type} (A : Set U) (B : Set V) (a : U) (b : V) :
(a, b) ∈ A ×s B ↔ a ∈ A ∧ b ∈ B := by rfl
```

To type the subscript s after \times , type \backslash_s . Thus, to type \times_s , you can type $\backslash\text{times}_s$ or $\backslash x\backslash_s$. Notice that in the `notation` command that introduces the symbol \times_s , we have used the number 75 in positions where we used 50 when defining the notation \sim . Without going into detail about exactly what the three occurrences of 50 and 75 mean, we will just say that this tells Lean that \times_s is to be given higher precedence than \sim , and as a result an expression like $A \sim B \times_s C$ will be interpreted as $A \sim (B \times_s C)$ rather than $(A \sim B) \times_s C$.

According to this definition, if A has type `Set U` and B has type `Set V`, then $A \times_s B$ has type `Set (U × V)`, and therefore $\uparrow(A \times_s B)$ is a subtype of $U \times V$. There is an obvious correspondence between $\uparrow(A \times_s B)$ and $\uparrow A \times \uparrow B$ that can be used to prove that they are equinumerous:

```
lemma elt_set_prod {U V : Type} {A : Set U} {B : Set V} (p : ↑A × ↑B) :
(p.1.val, p.2.val) ∈ A ×s B := And.intro p.1.property p.2.property

def prod_type_to_prod_set {U V : Type}
(A : Set U) (B : Set V) (p : ↑A × ↑B) : ↑(A ×s B) :=
Subtype_elt (elt_set_prod p)

def prod_set_to_prod_type {U V : Type}
(A : Set U) (B : Set V) (p : ↑(A ×s B)) : ↑A × ↑B :=
(Subtype_elt p.property.left, Subtype_elt p.property.right)

lemma set_prod_equinum_type_prod {U V : Type} (A : Set U) (B : Set V) :
↑(A ×s B) ~ (↑A × ↑B) := by
```

```

set F : ↑(A ×s B) → ↑A × ↑B := prod_set_to_prod_type A B
set G : ↑A × ↑B → ↑(A ×s B) := prod_type_to_prod_set A B
have h1 : F ∘ G = id := by rfl
have h2 : G ∘ F = id := by rfl
have h3 : one_to_one F := Theorem_5_3_3_1 F G h2
have h4 : onto F := Theorem_5_3_3_2 F G h1
show ↑(A ×s B) ~ (↑A × ↑B) from Exists.intro F (And.intro h3 h4)
done

```

Using this lemma we can now prove a more convenient set version of the first part of Theorem 8.1.2.

```

theorem Theorem_8_1_2_1_set
{U V W X : Type} {A : Set U} {B : Set V} {C : Set W} {D : Set X}
(h1 : A ~ B) (h2 : C ~ D) : A ×s C ~ B ×s D := by
have h3 : ↑(A ×s C) ~ (↑A × ↑C) := set_prod_equinum_type_prod A C
have h4 : (↑A × ↑C) ~ (↑B × ↑D) := Theorem_8_1_2_1_type h1 h2
have h5 : ↑(B ×s D) ~ (↑B × ↑D) := set_prod_equinum_type_prod B D
have h6 : (↑B × ↑D) ~ ↑(B ×s D) := Theorem_8_1_3_2 h5
have h7 : ↑(A ×s C) ~ (↑B × ↑D) := Theorem_8_1_3_3 h3 h4
show ↑(A ×s C) ~ ↑(B ×s D) from Theorem_8_1_3_3 h7 h6
done

```

As explained in Section 7.4 of *HTPI*, a key fact used in the proof of Theorem 7.4.4 is that if A is a set with m elements and B is a set with n elements, then $A \times B$ has mn elements. Section 7.4 of *HTPI* gives an intuitive explanation of this fact, but we'll need to prove it in Lean. In other words, we need to prove the following theorem:

```

theorem numElts_prod {U V : Type} {A : Set U} {B : Set V} {m n : Nat}
(h1 : numElts A m) (h2 : numElts B n) : numElts (A ×s B) (m * n)

```

Here's our plan for this proof: The hypotheses `numElts A m` and `numElts B n` mean $I\ m \sim A$ and $I\ n \sim B$. Applying `Theorem_8_1_2_1_set` to these hypotheses, we can infer $I\ m \times_s I\ n \sim A \times_s B$. If we can prove that $I\ (m * n) \sim I\ m \times_s I\ n$, then we'll be able to conclude $I\ (m * n) \sim A \times_s B$, or in other words `numElts (A ×s B) (m * n)`, as required. Thus, the key to the proof is to show that $I\ (m * n) \sim I\ m \times_s I\ n$.

To prove this, we'll define a function from `Nat` to `Nat × Nat` that maps $I\ (m * n)$ to $I\ m \times_s I\ n$. The function we will use maps a natural number a to the quotient and remainder when a is divided by n .

```

def qr (n a : Nat) : Nat × Nat := (a / n, a % n)

lemma qr_def (n a : Nat) : qr n a = (a / n, a % n) := by rfl

lemma qr_one_one (n : Nat) : one_to_one (qr n) := by
  define
    fix a1 : Nat; fix a2 : Nat
  assume h1 : qr n a1 = qr n a2           --Goal : a1 = a2
  rewrite [qr_def, qr_def] at h1
  have h2 : a1 / n = a2 / n ∧ a1 % n = a2 % n := Prod.mk.inj h1
  show a1 = a2 from
    calc a1
      _ = n * (a1 / n) + a1 % n := (Nat.div_add_mod a1 n).symm
      _ = n * (a2 / n) + a2 % n := by rw [h2.left, h2.right]
      _ = a2 := Nat.div_add_mod a2 n
  done

lemma qr_image (m n : Nat) :
  image (qr n) (I (m * n)) = (I m) ×s (I n) := sorry

lemma I_prod (m n : Nat) : I (m * n) ~ I m ×s I n := by
  rewrite [←qr_image m n]
  show I (m * n) ~ image (qr n) (I (m * n)) from
    equinum_image (one_one_on_of_one_one (qr_one_one n) (I (m * n)))
  done

theorem numElts_prod {U V : Type} {A : Set U} {B : Set V} {m n : Nat}
  (h1 : numElts A m) (h2 : numElts B n) : numElts (A ×s B) (m * n) := by
  rewrite [numElts_def] at h1      --h1 : I m ~ A
  rewrite [numElts_def] at h2      --h2 : I n ~ B
  rewrite [numElts_def]          --Goal : I (m * n) ~ A ×s B
  have h3 : I m ×s I n ~ A ×s B := Theorem_8_1_2_1_set h1 h2
  have h4 : I (m * n) ~ I m ×s I n := I_prod m n
  show I (m * n) ~ A ×s B from Theorem_8_1_3_3 h4 h3
  done

```

Our strategy for proving Theorem 7.4.4 will be to show that if m and n are relatively prime, then $\text{set_rp_below}(m * n) \sim \text{set_rp_below } m \times_s \text{set_rp_below } n$. Once again, we use a function from Nat to $\text{Nat} \times \text{Nat}$ to show that these sets are equinumerous. This time, the function will map a to $(a \% m, a \% n)$.

```
def mod_mod (m n a : Nat) : Nat × Nat := (a % m, a % n)

lemma mod_mod_def (m n a : Nat) : mod_mod m n a = (a % m, a % n) := by rfl
```

Our proof will make use of several theorems from the exercises of Sections 7.3 and 7.4:

```
theorem congr_rel_prime {m a b : Nat} (h1 : a ≡ b (MOD m)) :
  rel_prime m a ↔ rel_prime m b := sorry

theorem rel_prime_mod (m a : Nat) :
  rel_prime m (a % m) ↔ rel_prime m a := sorry

theorem congr_iff_mod_eq_Nat (m a b : Nat) [NeZero m] :
  ↑a ≡ ↑b (MOD m) ↔ a % m = b % m := sorry

lemma Lemma_7_4_6 {a b c : Nat} :
  rel_prime (a * b) c ↔ rel_prime a c ∧ rel_prime b c := sorry
```

Combining these with other theorems from Chapter 7, we can now use `mod_mod m n` to show that `set_rp_below (m * n) ~ set_rp_below m ×s set_rp_below n`.

```
lemma left_NeZero_of_mul {m n : Nat} (h : m * n ≠ 0) : NeZero m :=
  neZero_iff.rtl (left_ne_zero_of_mul h)

lemma right_NeZero_of_mul {m n : Nat} (h : m * n ≠ 0) : NeZero n :=
  neZero_iff.rtl (right_ne_zero_of_mul h)

lemma mod_mod_one_one_on {m n : Nat} (h1 : rel_prime m n) :
  one_one_on (mod_mod m n) (set_rp_below (m * n)) := by
  define
  fix a1 : Nat; fix a2 : Nat
  assume h2 : a1 ∈ set_rp_below (m * n)
  assume h3 : a2 ∈ set_rp_below (m * n)
  assume h4 : mod_mod m n a1 = mod_mod m n a2 --Goal : a1 = a2
  define at h2; define at h3
  rewrite [mod_mod_def, mod_mod_def] at h4
  have h5 : a1 % m = a2 % m ∧ a1 % n = a2 % n := Prod.mk.inj h4
  have h6 : m * n ≠ 0 := by linarith
  have h7 : NeZero m := left_NeZero_of_mul h6
  have h8 : NeZero n := right_NeZero_of_mul h6
  rewrite [←congr_iff_mod_eq_Nat, ←congr_iff_mod_eq_Nat] at h5
```

```
--h5 : ↑a1 ≡ ↑a2 (MOD m) ∧ ↑a1 ≡ ↑a2 (MOD n)
rewrite [←Lemma_7_4_5 _ _ h1] at h5 --h5 : ↑a1 ≡ ↑a2 (MOD m * n)
rewrite [congr_if_mod_eq_Nat] at h5 --h5 : a1 % (m * n) = a2 % (m * n)
rewrite [Nat.mod_eq_of_lt h2.right, Nat.mod_eq_of_lt h3.right] at h5
show a1 = a2 from h5
done

lemma mod_elt_set_rp_below {a m : Nat} [NeZero m] (h1 : rel_prime m a) :
  a % m ∈ set_rp_below m := by
  define --Goal : rel_prime m (a % m) ∧ a % m < m
  rewrite [rel_prime_mod] --Goal : rel_prime m a ∧ a % m < m
  show rel_prime m a ∧ a % m < m from
    And.intro h1 (mod_nonzero_lt a (NeZero.ne m))
  done

lemma mod_mod_image {m n : Nat} (h1 : rel_prime m n) :
  image (mod_mod m n) (set_rp_below (m * n)) =
  (set_rp_below m) ×s (set_rp_below n) := by
  apply Set.ext
  fix (b, c) : Nat × Nat
  apply Iff.intro
  · -- (→)
    assume h2 : (b, c) ∈ image (mod_mod m n) (set_rp_below (m * n))
    define at h2
    obtain (a : Nat)
      (h3 : a ∈ set_rp_below (m * n) ∧ mod_mod m n a = (b, c)) from h2
    rewrite [set_rp_below_def, mod_mod_def] at h3
    have h4 : rel_prime (m * n) a := h3.left.left
    rewrite [Lemma_7_4_6] at h4 --h4 : rel_prime m a ∧ rel_prime n a
    have h5 : a % m = b ∧ a % n = c := Prod.mk.inj h3.right
    define
    rewrite [←h5.left, ←h5.right]
      --Goal : a % m ∈ set_rp_below m ∧ a % n ∈ set_rp_below n
    have h6 : m * n ≠ 0 := by linarith
    have h7 : NeZero m := left_NeZero_of_mul h6
    have h8 : NeZero n := right_NeZero_of_mul h6
    apply And.intro
    · -- Proof that a % m ∈ set_rp_below m
      show a % m ∈ set_rp_below m from mod_elt_set_rp_below h4.left
      done
    · -- Proof that a % n ∈ set_rp_below n
      show a % n ∈ set_rp_below n from mod_elt_set_rp_below h4.right
```

```

done
done
· -- (→)
assume h2 : (b, c) ∈ set_rp_below m ×s set_rp_below n
rewrite [set_prod_def, set_rp_below_def, set_rp_below_def] at h2
  --h2 : (rel_prime m b ∧ b < m) ∧ (rel_prime n c ∧ c < n)
define
have h3 : m ≠ 0 := by linarith
have h4 : n ≠ 0 := by linarith
rewrite [←neZero_iff] at h3
rewrite [←neZero_iff] at h4
obtain (a : Nat) (h5 : a < m * n ∧ a ≡ b (MOD m) ∧ a ≡ c (MOD n))
  from Lemma_7_4_7 h1 b c
apply Exists.intro a
apply And.intro
· -- Proof of a ∈ set_rp_below (m * n)
define           --Goal : rel_prime (m * n) a ∧ a < m * n
apply And.intro _ h5.left
rewrite [Lemma_7_4_6]   --Goal : rel_prime m a ∧ rel_prime n a
rewrite [congr_rel_prime h5.right.left,
  congr_rel_prime h5.right.right]
show rel_prime m b ∧ rel_prime n c from
  And.intro h2.left.left h2.right.left
done
· -- Proof of mod_mod m n a = (b, c)
rewrite [congr_iff_mod_eq_Nat, congr_iff_mod_eq_Nat] at h5
rewrite [mod_mod_def, h5.right.left, h5.right.right]
  --Goal : (b % m, c % n) = (b, c)
rewrite [Nat.mod_eq_of_lt h2.left.right,
  Nat.mod_eq_of_lt h2.right.right]
rfl
done
done
done

lemma set_rp_below_prod {m n : Nat} (h1 : rel_prime m n) :
  set_rp_below (m * n) ~ (set_rp_below m) ×s (set_rp_below n) := by
  rewrite [←mod_mod_image h1]
  show set_rp_below (m * n) ~
    image (mod_mod m n) (set_rp_below (m * n)) from
    equinum_image (mod_mod_one_one_on h1)
done

```

We finally have everything we need to prove Theorem 7.4.4.

```
lemma eq_numElts_of_equinum {U V : Type} {A : Set U} {B : Set V} {n : Nat}
  (h1 : A ~ B) (h2 : numElts A n) : numElts B n := by
  rewrite [numElts_def] at h2 --h2 : I n ~ A
  rewrite [numElts_def] --Goal : I n ~ B
  show I n ~ B from Theorem_8_1_3_3 h2 h1
  done

theorem Theorem_7_4_4 {m n : Nat} (h1 : rel_prime m n) :
  phi (m * n) = (phi m) * (phi n) := by
  have h2 : numElts (set_rp_below m) (phi m) := phi_is_numElts m
  have h3 : numElts (set_rp_below n) (phi n) := phi_is_numElts n
  have h4 : numElts (set_rp_below (m * n)) (phi (m * n)) :=
    phi_is_numElts (m * n)
  have h5 : numElts (set_rp_below m ×s set_rp_below n) (phi (m * n)) :=
    eq_numElts_of_equinum (set_rp_below_prod h1) h4
  have h6 : numElts (set_rp_below m ×s set_rp_below n) (phi m * phi n) :=
    numElts_prod h2 h3
  show phi (m * n) = phi m * phi n from Exercise_8_1_6b h5 h6
  done
```

Exercises

1.

```
lemma image_empty {U : Type} {A : Set U}
  (f : U → Nat) (h : empty A) : image f A = I 0 := sorry
```
2.

```
lemma remove_one_equinum
  {U V : Type} {A : Set U} {B : Set V} {a : U} {b : V} {f : U → V}
  (h1 : one_one_on f A) (h2 : image f A = B)
  (h3 : a ∈ A) (h4 : f a = b) : ↑(A \ {a}) ~ ↑(B \ {b}) := sorry
```
3.

```
lemma singleton_of_diff_empty {U : Type} {A : Set U} {a : U}
  (h1 : a ∈ A) (h2 : empty (A \ {a})) : A = {a} := sorry
```
4.

```
lemma eq_zero_of_I_zero_equinum {n : Nat} (h : I 0 ~ I n) : n = 0 := sorry
```
5.

```
--Hint: Use mathematical induction.
theorem Exercise_8_1_6a : ∀ {m n : Nat}, (I m ~ I n) → m = n := sorry
```
6.

```
theorem Exercise_8_1_6b {U : Type} {A : Set U} {m n : Nat}
  (h1 : numElts A m) (h2 : numElts A n) : m = n := sorry
```

7.

```
lemma neb_nrp (m : Nat) : ∀ {k : Nat}, k ≤ m →
    num_elts_below (set_rp_below m) k = num_rp_below m k := sorry
```
8.

```
--Hint: You might find it helpful to apply the theorem div_mod_char
--from the exercises of Section 6.4.
lemma qr_image (m n : Nat) :
    image (qr n) (I (m * n)) = I m ×s I n := sorry
```

Suppose U and V are types, A and C have type $\text{Set } U$, and we have two functions $f : A \rightarrow V$ and $g : C \rightarrow V$. Then we can define a new function $\text{func_union } f g : A \cup C \rightarrow V$ as follows:

```
lemma is_elt_snd_of_not_fst {U : Type} {A C : Set U} {x : U}
  (h1 : x ∈ A ∪ C) (h2 : x ∉ A) : x ∈ C := by
  disj_syll h1 h2
  show x ∈ C from h1
  done

def elt_snd_of_not_fst {U : Type} {A C : Set U} {x : ↑(A ∪ C)}
  (h : x.val ∉ A) : C :=
  Subtype_elt (is_elt_snd_of_not_fst x.property h)

noncomputable def func_union {U V : Type} {A C : Set U}
  (f : A → V) (g : C → V) (x : ↑(A ∪ C)) : V :=
  if test : x.val ∈ A then f (Subtype_elt test)
  else g (elt_snd_of_not_fst test)
```

Note that in the definition of func_union , we have $\text{test} : x.\text{val} \in A$ in the `then` clause and $\text{test} : x.\text{val} \notin A$ in the `else` clause. If $x.\text{val} \in A$ then the value of $\text{func_union } f g x$ is determined by f , and if $x.\text{val} \notin A$ then it is determined by g . The next two exercises ask you to prove properties of this function

9.

```
lemma func_union_one_one {U V : Type} {A C : Set U}
    {f : A → V} {g : C → V} (h1 : empty (range f ∩ range g))
    (h2 : one_to_one f) (h3 : one_to_one g) :
    one_to_one (func_union f g) := sorry
```
10.

```
lemma func_union_range {U V : Type} {A C : Set U}
    (f : A → V) (g : C → V) (h : empty (A ∩ C)) :
    range (func_union f g) = range f ∪ range g := sorry
```

11. --Hint: Use the last two exercises.

```
theorem Theorem_8_1_2_2
  {U V : Type} {A C : Set U} {B D : Set V}
  (h1 : empty (A ∩ C)) (h2 : empty (B ∩ D))
  (h3 : A ~ B) (h4 : C ~ D) : ↑(A ∪ C) ~ ↑(B ∪ D) := sorry
```
12. lemma shift_I_equinum (n m : Nat) : I m ~ ↑(I (n + m) \ I n) := sorry
13. theorem Theorem_8_1_7 {U : Type} {A B : Set U} {n m : Nat}
 (h1 : empty (A ∩ B)) (h2 : numElts A n) (h3 : numElts B m) :
 numElts (A ∪ B) (n + m) := sorry
14. theorem equinum_sub {U V : Type} {A C : Set U} {B : Set V}
 (h1 : A ~ B) (h2 : C ⊆ A) : ∃ (D : Set V), D ⊆ B ∧ C ~ D := sorry
15. theorem Exercise_8_1_8b {U : Type} {A B : Set U}
 (h1 : finite A) (h2 : B ⊆ A) : finite B := sorry
16. theorem finite_bdd {A : Set Nat} (h : finite A) :
 ∃ (m : Nat), ∀ n ∈ A, n < m := sorry
17. lemma N_not_finite : ¬finite Nat := sorry
18. theorem denum_not_finite (U : Type)
 (h : denum U) : ¬finite U := sorry
19. --Hint: Use Like_Exercise_6_2_16 from the exercises of Section 6.2.
 theorem Exercise_6_2_16 {U : Type} {f : U → U}
 (h1 : one_to_one f) (h2 : finite U) : onto f := sorry

8.2. Countable and Uncountable Sets

Section 8.2 of *HTPI* shows that many set-theoretic operations, when applied to countable sets, produce results that are countable. For example, the first part of Theorem 8.2.1 shows that a Cartesian product of countable sets is countable. Our proof of this statement in Lean is based on `Theorem_8_1_2_1_set` and the denumerability of `Nat × Nat`. We also use an exercise from Section 8.1.

```
theorem NxN_denum : denum (Nat × Nat) := Theorem_8_1_3_2 NxN_equinum_N

theorem Theorem_8_2_1_1 {U V : Type} {A : Set U} {B : Set V}
  (h1 : ctable A) (h2 : ctable B) : ctable (A ×s B) := by
  rewrite [ctble_iff_set_equinum] at h1
```

```

rewrite [ctble_iff_set_nat_equinum] at h2
obtain (J : Set Nat) (h3 : J ~ A) from h1
obtain (K : Set Nat) (h4 : K ~ B) from h2
have h5 : J ×s K ~ A ×s B := Theorem_8_1_2_1_set h3 h4
have h6 : ctable (Nat × Nat) := Or.inr NxN_denum
have h7 : ctable (J ×s K) := ctable_set_of_ctble_type h6 (J ×s K)
show ctable (A ×s B) from ctable_of_ctble_equinum h5 h7
done

```

The second part of Theorem 8.2.1 shows that a union of two countable sets is countable, but, as we ask you to show in the exercises, it is superseded by Theorem 8.2.2, which says that a union of a countable family of countable sets is countable. So we will skip ahead to Theorem 8.2.2. Here's how we state the theorem in Lean:

```

theorem Theorem_8_2_2 {U : Type} {F : Set (Set U)}
  (h1 : ctable F) (h2 : ∀ A ∈ F, ctable A) : ctable (U₀ F)

```

As in the proof in *HTPI*, we will use the characterization of countability from [Theorem_8_1_5_2](#). We first consider the case in which F is nonempty and also all elements of F are nonempty. According to [Theorem_8_1_5_2](#), the hypotheses $h1$ and $h2$ then imply that there is a function $j : \text{Nat} \rightarrow \text{Set } U$ such that $F \subseteq \text{range } j$, and also for each $A \in F$ there is a function $g_A : \text{Nat} \rightarrow U$ with $A \subseteq \text{range } g_A$. We begin by proving an easier version of the theorem, where we assume that we have a function g from $\text{Set } U$ to $\text{Nat} \rightarrow U$ that supplies, for each $A \in F$, the required function g_A . Imitating the proof in *HTPI*, we can use j and g to construct the function needed to prove that $U_0 F$ is countable.

```

lemma Lemma_8_2_2_1 {U : Type} {F : Set (Set U)} {g : Set U → Nat → U}
  (h1 : ctable F) (h2 : ¬empty F) (h3 : ∀ A ∈ F, A ⊆ range (g A)) :
  ctable (U₀ F) := by
  rewrite [Theorem_8_1_5_2] at h1
  disj_syll h1 h2           --h1 : ∃ (f : Nat → Set U), F ⊆ range f
  rewrite [Theorem_8_1_5_2]
  apply Or.inr               --Goal : ∃ (f : Nat → Set U), U₀F ⊆ range f
  obtain (j : Nat → Set U) (h4 : F ⊆ range j) from h1
  obtain (p : Nat → Nat × Nat) (h5 : one_to_one p ∧ onto p) from NxN_denum
  set f : Nat → U := fun (n : Nat) => g (j (p n).1) (p n).2
  apply Exists.intro f
  fix x : U
  assume h6 : x ∈ U₀ F
  obtain (A : Set U) (h7 : A ∈ F ∧ x ∈ A) from h6
  obtain (n1 : Nat) (h8 : j n1 = A) from h4 h7.left

```

```

obtain (n2 : Nat) (h9 : g A n2 = x) from h3 A h7.left h7.right
obtain (n : Nat) (h10 : p n = (n1, n2)) from h5.right (n1, n2)
apply Exists.intro n
show f n = x from
calc f n
  _ = g (j (p n).1) (p n).2 := by rfl
  _ = g (j n1) n2 := by rw [h10]
  _ = g A n2 := by rw [h8]
  _ = x := by rw [h9]
done

```

How can we use `Lemma_8_2_2_1` to prove `Theorem_8_2_2` (still assuming that F and every element of F are nonempty)? We must use the hypothesis $h2 : \forall A \in F, \text{ctble } A$ in `Theorem_8_2_2` to produce the function g required in `Lemma_8_2_2_1`. As we have already observed, `Theorem_8_1_5_2` guarantees that for each $A \in F$, an appropriate function $gA : \text{Nat} \rightarrow U$ exists. We need a function g that will *choose* such a function gA for each A . A function with this property is often called a *choice function*. And now we come to a delicate point that was skipped over in *HTPI*: to prove the existence of a choice function, we must use a statement called the *axiom of choice*.¹

The distinction between the existence of an appropriate function gA for each A and the existence of a function that chooses such a function for each A is a subtle one. Perhaps for this reason, many people find the axiom of choice to be intuitively obvious. *HTPI* took advantage of this intuition to skip over this step in the proof without comment. But of course Lean won't let us skip anything!

To implement the axiom of choice, Lean uses a function called `Classical.choose`. Given a proof h of a statement of the form $\exists (x : U), P x$, the expression `Classical.choose h` produces (“chooses”) some $u : U$ such that $P u$ is true. There is also a theorem `Classical.choose_spec` that guarantees that the `Classical.choose` function meets its specification—that is, $P (\text{Classical.choose } h)$ is true. Using these, we can prove a lemma that will bridge the gap between `Lemma_8_2_2_1` and `Theorem_8_2_2`.

```

lemma Lemma_8_2_2_2 {U : Type} {F : Set (Set U)} (h1 : \forall A \in F, \text{ctble } A)
  (h2 : \neg\text{empty } F) (h3 : \forall A \in F, \neg\text{empty } A):
  \exists (g : Set U \rightarrow \text{Nat} \rightarrow U), \forall A \in F, A \subseteq \text{range } (g A) := by
  have h4 : \forall (A : Set U), \exists (gA : \text{Nat} \rightarrow U),
    A \in F \rightarrow A \subseteq \text{range } gA := by
  fix A : Set U
  by_cases h4 : A \in F
  · -- Case 1. h4 : A \in F

```

¹The axiom of choice was first stated by Ernst Zermelo in 1904. You can learn more about it in Gregory H. Moore, *Zermelo's Axiom of Choice: Its Origins, Development, and Influence*, Dover Publications, 2013. See also https://en.wikipedia.org/wiki/Axiom_of_choice.

```

have h5 : ctbl A := h1 A h4
rewrite [Theorem_8_1_5_2] at h5
disj_syll h5 (h3 A h4) --h5 : ∃ (f : Nat → U), A ⊆ range f
obtain (gA : Nat → U) (h6 : A ⊆ range gA) from h5
apply Exists.intro gA
assume h7 : A ∈ F
show A ⊆ range gA from h6
done
• -- Case 2. h4 : A ∉ F
rewrite [not_empty_iff_exists_elt] at h2
obtain (A0 : Set U) (h5 : A0 ∈ F) from h2
have h6 : ¬empty A0 := h3 A0 h5
rewrite [not_empty_iff_exists_elt] at h6
obtain (x0 : U) (h7 : x0 ∈ A0) from h6
set gA : Nat → U := constant_func Nat x0
apply Exists.intro gA
contrapos
assume h8 : A ∉ range gA
show A ∉ F from h4
done
done
set g : Set U → Nat → U := fun (A : Set U) => Classical.choose (h4 A)
apply Exists.intro g
fix A : Set U
show A ∈ F → A ⊆ range (g A) from Classical.choose_spec (h4 A)
done

```

Notice that the domain of the function g in `Lemma_8_2_2_2` is `Set U`, not `F`. Thus, we must produce a function gA for every $A : \text{Set } U$, but it is only when $A \in F$ that we care what gA is. Thus, the proof above just picks a default value (`constant_func Nat x0`) when $A \notin F$.

We can now prove the theorem, still under the assumption that all elements of F are nonempty. If F is empty, then we can show that $\cup_0 F$ is empty, so it has zero elements, which implies that it is finite and therefore countable. If F is not empty, then we can combine `Lemma_8_2_2_1` and `Lemma_8_2_2_2` to prove the theorem.

```

lemma Lemma_8_2_2_3 {U : Type} {F : Set (Set U)}
  (h1 : ctbl F) (h2 : ∀ A ∈ F, ctbl A) (h3 : ∀ A ∈ F, ¬empty A) :
  ctbl (U0 F) := by
  by_cases h4 : empty F
  • -- Case 1. h4 : empty F
    have h5 : empty (U0 F) := by

```

```

contradict h4 with h5
rewrite [not_empty_iff_exists_elt] at h5
obtain (x : U) (h6 : x ∈ U₀ F) from h5
obtain (A : Set U) (h7 : A ∈ F ∧ x ∈ A) from h6
show ∃ (x : Set U), x ∈ F from Exists.intro A h7.left
done
rewrite [ $\leftarrow$ zero_elts_iff_empty] at h5      --h5 : numElts (U₀ F) 0
define
apply Or.inl
rewrite [finite_def]
show ∃ (n : Nat), numElts (U₀ F) n from Exists.intro 0 h5
done
• -- Case 2. h4 :  $\neg$ empty F
obtain (g : Set U → Nat → U) (h5 : ∀ A ∈ F, A ⊆ range (g A)) from
Lemma_8_2_2_2 h2 h4 h3
show ctable (U₀ F) from Lemma_8_2_2_1 h1 h4 h5
done

```

Finally, we deal with the possibility that F contains the empty set. As in *HTPI*, we show that we can simply remove the empty set from F and then apply our earlier reasoning.

```

lemma remove_empty_subset {U : Type} (F : Set (Set U)) :
{A : Set U | A ∈ F ∧  $\neg$ empty A} ⊆ F := by
fix X : Set U
assume h1 : X ∈ {A : Set U | A ∈ F ∧  $\neg$ empty A}
define at h1
show X ∈ F from h1.left
done

lemma remove_empty_union_eq {U : Type} (F : Set (Set U)) :
U₀ {A : Set U | A ∈ F ∧  $\neg$ empty A} = U₀ F := sorry

theorem Theorem_8_2_2 {U : Type} {F : Set (Set U)}
(h1 : ctable F) (h2 : ∀ A ∈ F, ctable A) : ctable (U₀ F) := by
set G : Set (Set U) := {A : Set U | A ∈ F ∧  $\neg$ empty A}
have h3 : G ⊆ F := remove_empty_subset F
have h4 : U₀ G = U₀ F := remove_empty_union_eq F
rewrite [ $\leftarrow$ h4]
have h5 : ctable G := Exercise_8_1_17 h3 h1
have h6 : ∀ A ∈ G, ctable A := by
fix A : Set U
assume h6 : A ∈ G

```

```

show ctable A from h2 A (h3 h6)
done
have h7 : ∀ A ∈ G, ¬empty A := by
  fix A : Set U
  assume h7 : A ∈ G
  define at h7
  show ¬empty A from h7.right
  done
show ctable (U₀ G) from Lemma_8_2_2_3 h5 h6 h7
done

```

By the way, we can now explain a mystery from Section 5.1. The reason we skipped the proof of the right-to-left direction of `func_from_graph` is that the proof uses `Classical.choose` and `Classical.choose_spec`. Now that you know about this function and theorem, we can show you the proof.

```

theorem func_from_graph_rtl {A B : Type} (F : Set (A × B)) :
  is_func_graph F → (∃ (f : A → B), graph f = F) := by
  assume h1 : is_func_graph F
  define at h1    --h1 : ∀ (x : A), ∃! (y : B), (x, y) ∈ F
  have h2 : ∀ (x : A), ∃ (y : B), (x, y) ∈ F := by
    fix x : A
    obtain (y : B) (h3 : (x, y) ∈ F)
      (h4 : ∀ (y1 y2 : B), (x, y1) ∈ F → (x, y2) ∈ F → y1 = y2) from h1 x
    show ∃ (y : B), (x, y) ∈ F from Exists.intro y h3
    done
  set f : A → B := fun (x : A) => Classical.choose (h2 x)
  apply Exists.intro f
  apply Set.ext
  fix (x, y) : A × B
  have h3 : (x, f x) ∈ F := Classical.choose_spec (h2 x)
  apply Iff.intro
  · -- (→)
    assume h4 : (x, y) ∈ graph f
    define at h4    --h4 : f x = y
    rewrite [h4] at h3
    show (x, y) ∈ F from h3
    done
  · -- (←)
    assume h4 : (x, y) ∈ F
    define          --Goal : f x = y
    obtain (z : B) (h5 : (x, z) ∈ F)

```

8.2. Countable and Uncountable Sets

```
(h6 : ∀ (y1 y2 : B), (x, y1) ∈ F → (x, y2) ∈ F → y1 = y2) from h1 x
show f x = y from h6 (f x) y h3 h4
done
done
```

There is one more theorem in Section 8.2 of *HTPI* showing that a set-theoretic operation, when applied to a countable set, gives a countable result. Theorem 8.2.4 says that if a set A is countable, then the set of all finite sequences of elements of A is also countable. In *HTPI*, finite sequences are represented by functions, but in Lean it is easier to use lists. Thus, if A has type `Set U`, then we define a *finite sequence of elements of A* to be a list $l : List U$ with the property that every entry of l is an element of A . Letting `seq A` denote the set of all finite sequences of elements of A , our version of Theorem 8.2.4 will say that if A is countable, then so is `seq A`.

```
def seq {U : Type} (A : Set U) : Set (List U) :=
{ l : List U | ∀ x ∈ l, x ∈ A }

lemma seq_def {U : Type} (A : Set U) (l : List U) :
l ∈ seq A ↔ ∀ x ∈ l, x ∈ A := by rfl

theorem Theorem_8_2_4 {U : Type} {A : Set U}
(h1 : ctble A) : ctble (seq A)
```

Our proof of `Theorem_8_2_4` will use exactly the same strategy as the proof in *HTPI*. We begin by showing that, for every natural number n , the set of sequences of elements of A of length n is countable. The proof is by mathematical induction. The base case is easy, because the only sequence of length 0 is the `nil` list.

```
def seq_by_length {U : Type} (A : Set U) (n : Nat) : Set (List U) :=
{ l : List U | l ∈ seq A ∧ l.length = n }

lemma sbl_base {U : Type} (A : Set U) : seq_by_length A 0 = {[]} := by
apply Set.ext
fix l : List U
apply Iff.intro
· -- (→)
  assume h1 : l ∈ seq_by_length A 0
  define at h1  --h1 : l ∈ seq A ∧ l.length = 0
  rewrite [List.length_eq_zero_iff] at h1
  define
  show l = [] from h1.right
```

```

done
· -- (→)
assume h1 : l ∈ {[]}
define at h1      --h1 : l = []
define          --Goal : l ∈ seq A ∧ l.length = 0
apply And.intro _ (List.length_eq_zero_iff.rtl h1)
define          --Goal : ∀ x ∈ l, x ∈ A
fix x : U
contrapos
assume h2 : x ∉ A
rewrite [h1]
show x ∉ [] from List.not_mem_nil
done
done

```

For the induction step, the key idea is that $A \times_s (\text{seq_by_length } A n) \sim \text{seq_by_length } A (n + 1)$. To prove this, we define a function `seq_cons` U that matches up $A \times_s (\text{seq_by_length } A n)$ with $\text{seq_by_length } A (n + 1)$.

```

def seq_cons (U : Type) (p : U × (List U)) : List U := p.1 :: p.2

lemma seq_cons_def {U : Type} (x : U) (l : List U) :
  seq_cons U (x, l) = x :: l := by rfl

lemma seq_cons_one_one (U : Type) : one_to_one (seq_cons U) := by
  fix (a1, l1) : U × List U; fix (a2, l2) : U × List U
  assume h1 : seq_cons U (a1, l1) = seq_cons U (a2, l2)
  rewrite [seq_cons_def, seq_cons_def] at h1 --h1 : a1 :: l1 = a2 :: l2
  rewrite [List.cons_eq_cons] at h1           --h1 : a1 = a2 ∧ l1 = l2
  rewrite [h1.left, h1.right]
  rfl
done

lemma seq_cons_image {U : Type} (A : Set U) (n : Nat) :
  image (seq_cons U) (A ×s (seq_by_length A n)) =
  seq_by_length A (n + 1) := sorry

lemma Lemma_8_2_4_1 {U : Type} (A : Set U) (n : Nat) :
  A ×s (seq_by_length A n) ~ seq_by_length A (n + 1) := by
  rewrite [←seq_cons_image A n]
  show A ×s seq_by_length A n ~
    image (seq_cons U) (A ×s seq_by_length A n) from equinum_image

```

8.2. Countable and Uncountable Sets

```
(one_one_on_of_one_one (seq_cons_one_one U) (A ×s (seq_by_length A n)))
done
```

With this preparation, we can now use `singleton_one_elt` to justify the base case of our induction proof and `Theorem_8_2_1_1` for the induction step.

```
lemma Lemma_8_2_4_2 {U : Type} {A : Set U} (h1 : ctable A) :
  ∀ (n : Nat), ctable (seq_by_length A n) := by
  by_induc
  · -- Base Case
    rewrite [sbl_base]      --Goal : ctable {[]}
    define
    apply Or.inl           --Goal : finite {[]}
    rewrite [finite_def]
    apply Exists.intro 1   --Goal : numElts {[]} 1
    show numElts {[]} 1 from singleton_one_elt []
    done
  · -- Induction Step
    fix n : Nat
    assume ih : ctable (seq_by_length A n)
    have h2 : A ×s (seq_by_length A n) ~ seq_by_length A (n + 1) :=
      Lemma_8_2_4_1 A n
    have h3 : ctable (A ×s (seq_by_length A n)) := Theorem_8_2_1_1 h1 ih
    show ctable (seq_by_length A (n + 1)) from ctable_of_ctble_equinum h2 h3
    done
  done
```

Our next step is to show that the union of all of the sets `seq_by_length A n`, for $n : \text{Nat}$, is `seq A`.

```
def sbl_set {U : Type} (A : Set U) : Set (Set (List U)) :=
  {S : Set (List U) | ∃ (n : Nat), seq_by_length A n = S}

lemma Lemma_8_2_4_3 {U : Type} (A : Set U) : U₀ (sbl_set A) = seq A := by
  apply Set.ext
  fix l : List U
  apply Iff.intro
  · -- (→)
    assume h1 : l ∈ U₀ (sbl_set A)
    define at h1
    obtain (S : Set (List U)) (h2 : S ∈ sbl_set A ∧ l ∈ S) from h1
    have h3 : S ∈ sbl_set A := h2.left
```

```

define at h3
obtain (n : Nat) (h4 : seq_by_length A n = S) from h3
have h5 : l ∈ S := h2.right
rewrite [←h4] at h5
define at h5
show l ∈ seq A from h5.left
done
· -- (→)
assume h1 : l ∈ seq A
define
set n : Nat := l.length
apply Exists.intro (seq_by_length A n)
apply And.intro
· -- Proof of seq_by_length A n ∈ sbl_set A
define
apply Exists.intro n
rfl
done
· -- Proof of l ∈ seq_by_length A n
define
apply And.intro h1
rfl
done
done
done

```

Of course, `sbl_set A` is countable. The easiest way to prove this is to note that `seq_by_length A` is a function from `Nat` to `Set (List U)` whose range contains all of the sets in `sbl_set A`.

```

lemma Lemma_8_2_4_4 {U : Type} (A : Set U) : ctle (sbl_set A) := by
  rewrite [Theorem_8_1_5_2]
  apply Or.inr --Goal : ∃ (f : Nat → Set (List U)), sbl_set A ⊆ range f
  apply Exists.intro (seq_by_length A)
  fix S : Set (List U)
  assume h1 : S ∈ sbl_set A
  define at h1; define
  show ∃ (x : Nat), seq_by_length A x = S from h1
  done

```

We now have everything we need to prove `Theorem_8_2_4` as an application of `Theorem_8_2_2`.

```

theorem Theorem_8_2_4 {U : Type} {A : Set U}
  (h1 : ctable A) : ctable (seq A) := by
  set F : Set (Set (List U)) := sbl_set A
  have h2 : ctable F := Lemma_8_2_4_4 A
  have h3 : ∀ S ∈ F, ctable S := by
    fix S : Set (List U)
    assume h3 : S ∈ F
    define at h3
    obtain (n : Nat) (h4 : seq_by_length A n = S) from h3
    rewrite [←h4]
    show ctable (seq_by_length A n) from Lemma_8_2_4_2 h1 n
    done
  rewrite [←Lemma_8_2_4_3 A]
  show ctable (U₀ sbl_set A) from Theorem_8_2_2 h2 h3
  done

```

There is a set-theoretic operation that can produce an uncountable set from a countable set: the power set operation. *HTPI* demonstrates this by proving Cantor's theorem (Theorem 8.2.5), which says that $\mathcal{P}(\mathbb{Z}^+)$ is uncountable. The strategy for this proof is tricky; it involves defining a set D using a method called *diagonalization*. For an explanation of the motivation behind this strategy, see *HTPI*.

Here we will prove in Lean that the collection of all sets of natural numbers is uncountable. There is no need to use the power set operation for this, because we have a type, namely `Set Nat`, that contains all sets of natural numbers. So our Lean version of Cantor's theorem says that `Set Nat` is uncountable.

```

theorem Cantor's_theorem : ¬ctble (Set Nat) := by
  by_contra h1
  rewrite [ctble_iff_set_nat_equinum] at h1
  obtain (J : Set Nat) (h2 : J ~ Set Nat) from h1
  obtain (F : J → Set Nat) (h3 : one_to_one F ∧ onto F) from h2
  set f : Nat → Set Nat := func_extend F ∅
  set D : Set Nat := {n : Nat | n ∉ f n}
  obtain (nJ : J) (h4 : F nJ = D) from h3.right D
  set n : Nat := nJ.val
  have h5 : n ∈ D ↔ n ∉ f n := by rfl
  have h6 : f n = F nJ := fe_elt F ∅ nJ
  rewrite [h6, h4] at h5      --h5 : n ∈ D ↔ n ∉ D
  by_cases h7 : n ∈ D
  · -- Case 1. h7 : n ∈ D
    contradict h7

```

```

show n ∉ D from h5.ltr h7
done
• -- Case 2. h7 : n ∉ D
contradict h7
show n ∈ D from h5.rtl h7
done
done

```

As a consequence of Theorem 8.2.5, *HTPI* shows that \mathbb{R} is uncountable. The proof is not hard, but it requires facts about the decimal expansions of real numbers. Developing those facts in Lean would take us too far afield, so we will skip the proof.

Exercises

1. `lemma pair_ctble {U : Type}
 (a b : U) : ctble ↑({a, b} : Set U) := sorry`
2. `--Hint: Use the previous exercise and Theorem_8_2_2.
theorem Theorem_8_2_1_2 {U : Type} {A B : Set U}
 (h1 : ctble A) (h2 : ctble B) : ctble ↑(A ∪ B) := sorry`
3. `lemma remove_empty_union_eq {U : Type} (F : Set (Set U)) :
 U₀ {A : Set U | A ∈ F ∧ ¬empty A} = U₀ F := sorry`
4. `lemma seq_cons_image {U : Type} (A : Set U) (n : Nat) :
 image (seq_cons U) (A ×s (seq_by_length A n)) =
 seq_by_length A (n + 1) := sorry`
5. `--Hint: Apply Theorem_8_2_4 to the set Univ U.
theorem Theorem_8_2_4_type {U : Type}
 (h : ctble U) : ctble (List U) := sorry`
6. `def list_to_set (U : Type) (l : List U) : Set U := {x : U | x ∈ l}

lemma list_to_set_def (U : Type) (l : List U) (x : U) :
 x ∈ list_to_set U l ↔ x ∈ l := by rfl

--Hint: Use induction on the size of A.
lemma set_from_list {U : Type} {A : Set U} (h : finite A) :
 ∃ (l : List U), list_to_set U l = A := sorry`

7. `--Hint: Use the previous exercise and Theorem_8_2_4_type.`
`theorem Like_Exercise_8_2_4 (U : Type) (h : ctle U) :`
`ctble {X : Set U | finite X} := sorry`
8. `theorem Exercise_8_2_6b (U V W : Type) :`
`((U × V) → W) ~ (U → V → W) := sorry`
9. `theorem Like_Exercise_8_2_7 : ∃ (P : Set (Set Nat)),`
`partition P ∧ denum P ∧ ∀ X ∈ P, denum X := sorry`
10. `theorem unctbly_many_inf_set_nat :`
`¬ctble {X : Set Nat | ¬finite X} := sorry`
11. `theorem Exercise_8_2_8 {U : Type} {A B : Set U}`
`(h : empty (A ∩ B)) : ℙ (A ∪ B) ~ ℙ A ×s ℙ B := sorry`

8.3. The Cantor–Schröder–Bernstein Theorem

The final section of *HTPI* proves the Cantor–Schröder–Bernstein theorem. The theorem says that if we have two sets such that there is a one-to-one function from each set to the other, then the two sets are equinumerous. We will prove it in Lean for types, but of course we can apply it to sets as well by coercing the sets to subtypes.

```
theorem Cantor_Schroeder_Bernstein_theorem
{U V : Type} {f : U → V} {g : V → U}
(h1 : one_to_one f) (h2 : one_to_one g) : U ~ V
```

To prove the theorem, we must produce a one-to-one, onto function h from U to V . Imitating the proof in *HTPI*, we will do this by defining a set $X : \text{Set } U$ and then using f to determine the values of h on elements of the domain that belong to X and the inverse of g for those that don't. That is, the graph of h will be the set `csb_func_graph f g X` defined as follows:

```
def csb_func_graph {U V : Type}
(f : U → V) (g : V → U) (X : Set U) : Set (U × V) :=
{(x, y) : U × V | (x ∈ X ∧ f x = y) ∨ (x ∉ X ∧ g y = x)}
```

Is `csb_func_graph f g X` the graph of a function? It is not hard to show that it is, as long as $\forall (x : U), x \notin X \rightarrow x \in \text{range } g$. We first state lemmas spelling out the two cases in the definition of `csb_func_graph f g X`, leaving the proof of the second as an exercise for you.

```

lemma csb_func_graph_X {U V : Type} {X : Set U} {x : U}
  (f : U → V) (g : V → U) (h : x ∈ X) (y : V) :
  (x, y) ∈ csb_func_graph f g X ↔ f x = y := by
apply Iff.intro
· -- (→)
  assume h1 : (x, y) ∈ csb_func_graph f g X
  define at h1
  have h2 : ¬(x ∉ X ∧ g y = x) := by
    demorgan
    show x ∈ X ∨ g y ≠ x from Or.inl h
    done
  disj_syll h1 h2      --h1 : x ∈ X ∧ f x = y
  show f x = y from h1.right
  done
· -- (←)
  assume h1 : f x = y
  define
  apply Or.inl
  show x ∈ X ∧ f x = y from And.intro h h1
  done
done

lemma csb_func_graph_not_X {U V : Type} {X : Set U} {x : U}
  (f : U → V) (g : V → U) (h : x ∉ X) (y : V) :
  (x, y) ∈ csb_func_graph f g X ↔ g y = x := sorry

lemma csb_func_graph_is_func_graph {U V : Type} {g : V → U} {X : Set U}
  (f : U → V) (h1 : ∀ (x : U), x ∉ X → x ∈ range g) (h2 : one_to_one g) :
  is_func_graph (csb_func_graph f g X) := by
define
fix x : U
by_cases h3 : x ∈ X
· -- Case 1. h3 : x ∈ X
  exists_unique
  · -- Existence
    apply Exists.intro (f x)
    rewrite [csb_func_graph_X f g h3]
    rfl
    done
  · -- Uniqueness
    fix y1 : V; fix y2 : V
    assume h4 : (x, y1) ∈ csb_func_graph f g X
    
```

```

assume h5 : (x, y2) ∈ csb_func_graph f g X
rewrite [csb_func_graph_X f g h3] at h4 --h4 : f x = y1
rewrite [csb_func_graph_X f g h3] at h5 --h5 : f x = y2
rewrite [←h4, ←h5]
rfl
done
done
done
• -- Case 2. h3 : x ∉ X
exists_unique
• -- Existence
obtain (y : V) (h4 : g y = x) from h1 x h3
apply Exists.intro y
rewrite [csb_func_graph_not_X f g h3]
show g y = x from h4
done
• -- Uniqueness
fix y1 : V; fix y2 : V
assume h4 : (x, y1) ∈ csb_func_graph f g X
assume h5 : (x, y2) ∈ csb_func_graph f g X
rewrite [csb_func_graph_not_X f g h3] at h4 --h4 : g y1 = x
rewrite [csb_func_graph_not_X f g h3] at h5 --h5 : g y2 = x
rewrite [←h5] at h4
show y1 = y2 from h2 y1 y2 h4
done
done
done
done

```

Our plan is to define h to be the function whose graph is `csb_func_graph f g X`. With this definition, the value of $h x$ for any $x : U$ can be determined by a simple rule: if $x \in X$, then $h x = f x$, and if $x \notin X$, then $h x$ has the property that $g(h x) = x$:

```

lemma csb_func_X
{U V : Type} {f h : U → V} {g : V → U} {X : Set U} {x : U}
(h1 : graph h = csb_func_graph f g X) (h2 : x ∈ X) : h x = f x := by
rewrite [←graph_def, h1, csb_func_graph_X f g h2]
rfl
done

lemma csb_func_not_X
{U V : Type} {f h : U → V} {g : V → U} {X : Set U} {x : U}
(h1 : graph h = csb_func_graph f g X) (h2 : x ∉ X) : g (h x) = x := by
have h3 : (x, h x) ∈ graph h := by rfl

```

8.3. The Cantor–Schröder–Bernstein Theorem

```

rewrite [h1, csb_func_graph_not_X f g h2] at h3
show g (h x) = x from h3
done

```

We still have to say how X will be defined. Let $A_0 = \{x : U \mid x \notin \text{range } g\}$. To make sure that the condition $\forall (x : U), x \notin X \rightarrow x \in \text{range } g$ is satisfied, we will need to have $A_0 \subseteq X$. As explained in *HTPI*, we can now get a suitable set X by repeatedly taking the image of A_0 under $g \circ f$. Fortunately, we defined functions in Section 6.5 that do what we need: `rep_image` ($g \circ f$) $n A_0$ is the result of taking the image of A_0 under $g \circ f$ n times. That is, `rep_image` ($g \circ f$) $0 A_0 = A_0$, `rep_image` ($g \circ f$) $1 A_0 = \text{image } (g \circ f) A_0$, `rep_image` ($g \circ f$) $2 A_0 = \text{image } (g \circ f) (\text{image } (g \circ f) A_0)$, and so on. We will define X to be the union of all of the sets `rep_image` ($g \circ f$) $n A_0$, which is given by the function `cumul_image` ($g \circ f$) A_0 .

To prove that h is one-to-one, we will need to know that it cannot happen that $h x_1 = h x_2$, $x_1 \in X$, and $x_2 \notin X$. After proving this last lemma, we are ready to prove the Cantor–Schröder–Bernstein theorem.

```

lemma csb_X_of_X
  {U V : Type} {f h : U → V} {g : V → U} {A0 : Set U} {x1 x2 : U}
  (h1 : graph h = csb_func_graph f g (cumul_image (g ∘ f) A0))
  (h2 : h x1 = h x2) (h3 : x1 ∈ cumul_image (g ∘ f) A0) :
  x2 ∈ cumul_image (g ∘ f) A0 := by
  by_contra h4           --h4 : x2 ∉ cumul_image (g ∘ f) A0
  rewrite [csb_func_X h1 h3] at h2 --h2 : f x1 = h x2
  have h5 : (g ∘ f) x1 = x2 :=
  calc (g ∘ f) x1
    _ = g (f x1) := by rfl
    _ = g (h x2) := by rw [h2]
    _ = x2 := csb_func_not_X h1 h4
  obtain (n : Nat) (h6 : x1 ∈ rep_image (g ∘ f) n A0) from h3
  contradict h4           --Goal : x2 ∈ cumul_image (g ∘ f) A0
  apply Exists.intro (n + 1) --Goal : x2 ∈ rep_image (g ∘ f) (n + 1) A0
  rewrite [rep_image_step]
  apply Exists.intro x1
  show x1 ∈ rep_image (g ∘ f) n A0 ∧ (g ∘ f) x1 = x2 from
    And.intro h6 h5
  done

theorem Cantor_Schroeder_Bernstein_theorem
  {U V : Type} {f : U → V} {g : V → U}
  (h1 : one_to_one f) (h2 : one_to_one g) : U ~ V := by
  set A0 : Set U := {x : U | x ∉ range g}

```

```

set X : Set U := cumul_image (g ∘ f) A0
set H : Set (U × V) := csb_func_graph f g X
have h3 : ∀ (x : U), x ∉ X → x ∈ range g := by
  fix x : U
  contrapos
  assume h3 : x ∉ range g
  define
  apply Exists.intro 0
  rewrite [rep_image_base]
  show x ∈ A0 from h3
  done
have h4 : is_func_graph H := csb_func_graph_is_func_graph f h3 h2
rewrite [←func_from_graph] at h4
obtain (h : U → V) (h5 : graph h = H) from h4
apply Exists.intro h
apply And.intro
· -- proof that h is one-to-one
  fix x1 : U; fix x2 : U
  assume h6 : h x1 = h x2
  by_cases h7 : x1 ∈ X
  · -- Case 1. h7 : x1 ∈ X
    have h8 : x2 ∈ X := csb_X_of_X h5 h6 h7
    rewrite [csb_func_X h5 h7, csb_func_X h5 h8] at h6 --h6 : f x1 = f x2
    show x1 = x2 from h1 x1 x2 h6
    done
  · -- Case 2. h7 : x1 ∉ X
    have h8 : x2 ∉ X := by
      contradict h7 with h8
    show x1 ∈ X from csb_X_of_X h5 h6.symm h8
    done
    show x1 = x2 from
      calc x1
        _ = g (h x1) := (csb_func_not_X h5 h7).symm
        _ = g (h x2) := by rw [h6]
        _ = x2 := csb_func_not_X h5 h8
    done
  done
· -- proof that h is onto
  fix y : V
  by_cases h6 : g y ∈ X
  · -- Case 1. h6 : g y ∈ X
    define at h6

```

```

obtain (n : Nat) (h7 : g y ∈ rep_image (g ∘ f) n A0) from h6
have h8 : n ≠ 0 := by
  by_contra h8
  rewrite [h8, rep_image_base] at h7 --h7 : g y ∈ A0
  define at h7 --h7 : ¬∃ (x : V), g x = g y
  contradict h7
  apply Exists.intro y
  rfl
  done
obtain (k : Nat) (h9 : n = k + 1) from
  exists_eq_add_one_of_ne_zero h8
rewrite [h9, rep_image_step] at h7
obtain (x : U)
  (h10 : x ∈ rep_image (g ∘ f) k A0 ∧ (g ∘ f) x = g y) from h7
have h11 : g (f x) = g y := h10.right
have h12 : f x = y := h2 (f x) y h11
have h13 : x ∈ X := Exists.intro k h10.left
apply Exists.intro x
rewrite [csb_func_X h5 h13]
show f x = y from h12
done
• -- Case 2. h6 : g y ∉ X
apply Exists.intro (g y)
have h7 : g (h (g y)) = g y := csb_func_not_X h5 h6
show h (g y) = y from h2 (h (g y)) y h7
done
done
done

```

Exercises

1. `lemma csb_func_graph_not_X {U V : Type} {X : Set U} {x : U}
 (f : U → V) (g : V → U) (h : x ∉ X) (y : V) :
 (x, y) ∈ csb_func_graph f g X ↔ g y = x := sorry`
2. `theorem intervals_equinumerous :
 {x : Real | 0 < x ∧ x < 1} ~ {x : Real | 0 < x ∧ x ≤ 1} := sorry`
3. The following theorem could be thought of as an extensionality principle for relations. You may find it useful in later exercises. Hint for proof: First show that `extension R = extension`

8.3. The Cantor–Schröder–Bernstein Theorem

S , and then use the fact that R and S can be determined from `extension R` and `extension S` (see Section 4.3).

```
theorem relext {U V : Type} {R S : Rel U V}
  (h : ∀ (u : U) (v : V), R u v ↔ S u v) : R = S := sorry
```

The next six exercises lead up to a proof that the set of all equivalence relations on the natural numbers is equinumerous with the type `Set Nat`. These exercises use the following definitions:

```
def EqRel (U : Type) : Set (BinRel U) :=
{R : BinRel U | equiv_rel R}

def Part (U : Type) : Set (Set (Set U)) :=
{P : Set (Set U) | partition P}

def EqRelExt (U : Type) : Set (Set (U × U)) :=
{E : Set (U × U) | ∃ (R : BinRel U), equiv_rel R ∧ extension R = E}

def shift_and_zero (X : Set Nat) : Set Nat :=
{x + 2 | x ∈ X} ∪ {0}

def shift_and_zero_comp (X : Set Nat) : Set Nat :=
{n : Nat | n ∉ shift_and_zero X}

def saz_pair (X : Set Nat) : Set (Set Nat) :=
{shift_and_zero X, shift_and_zero_comp X}
```

4. `theorem EqRel_equinum_Part (U : Type) : EqRel U ~ Part U := sorry`
5. `theorem EqRel_equinum_EqRelExt (U : Type) :
EqRel U ~ EqRelExt U := sorry`
6. `theorem EqRel_Nat_to_Set_Nat :
∃ (f : EqRel Nat → Set Nat), one_to_one f := sorry`
7. `theorem saz_pair_part (X : Set Nat) : saz_pair X ∈ Part Nat := sorry`
8. `theorem Set_Nat_to_EqRel_Nat :
∃ (f : Set Nat → EqRel Nat), one_to_one f := sorry`
9. `theorem EqRel_Nat_equinum_Set_Nat : EqRel Nat ~ Set Nat := sorry`

Appendix

Tactics Used

Tactics marked with an asterisk (*) are defined in the file `HTPIDefs.lean` in the HTPI Lean Package that accompanies this book. They will not work without that file. The others are standard Lean tactics or are defined in Lean's mathematics library, `mathlib`.

Tactic	Where Introduced
<code>apply</code>	Sections 3.1 & 3.2
<code>apply?</code>	Section 3.6
<code>assume*</code>	Introduction to Lean: A First Example
<code>bicond_neg*</code>	Introduction to Lean: Tactic Mode
<code>by_cases</code>	Section 3.5
<code>by_cases on*</code>	Section 3.5
<code>by_contra</code>	Sections 3.1 & 3.2
<code>by_induc*</code>	Section 6.1
<code>by_strong_induc*</code>	Section 6.4
<code>conditional*</code>	Introduction to Lean: Tactic Mode
<code>contradict*</code>	Sections 3.1 & 3.2
<code>contrapos*</code>	Introduction to Lean: A First Example
<code>decide</code>	Section 6.1
<code>define*</code>	Introduction to Lean: Types
<code>demorgan*</code>	Introduction to Lean: Tactic Mode
<code>disj_syll*</code>	Section 3.5
<code>double_neg*</code>	Introduction to Lean: Tactic Mode
<code>exact</code>	Section 3.6
<code>exists_unique*</code>	Section 3.6
<code>fix*</code>	Section 3.3
<code>have</code>	Introduction to Lean: A First Example
<code>linarith</code>	Section 6.1
<code>obtain*</code>	Section 3.3
<code>or_left*</code>	Section 3.5
<code>or_right*</code>	Section 3.5
<code>push_neg</code>	Section 8.1
<code>quant_neg*</code>	Section 3.3

Tactic	Where Introduced
<code>rel</code>	Section 6.3
<code>rewrite</code>	Section 3.6
<code>rfl</code>	Section 3.7
<code>ring</code>	Section 3.7
<code>rw</code>	Section 3.7
<code>set</code>	Section 4.5
<code>show*</code>	Introduction to Lean: A First Example
<code>trivial</code>	Section 7.2

Summary of Proof Techniques in Lean

To Prove a Goal of the Form ...

$\neg P$

The tactic `by_contra h` will initiate a proof by contradiction by introducing the new given `h : P` and setting the goal to be `False`. (See also the tactics for reexpressing negative statements under the [Reexpressing Statements](#) heading below. There is also further information about [proof by contradiction](#) below.)

$P \rightarrow Q$

The tactic `assume h : P` will introduce the new given `h : P` and set the goal to be `Q`. (See also the `contrapos` tactic under the [Reexpressing Statements](#) heading below.)

$P \wedge Q$

If you have `h1 : P` and `h2 : Q`, then `And.intro h1 h2` is a proof of `P \wedge Q`. If you have `h : P`, then the tactic `apply And.intro h` will set the goal to be `Q`, and if you have `h : Q`, then `apply And.intro _ h` will set the goal to be `P`. If you don't already have a proof of either `P` or `Q`, then `apply And.intro` will set `P` and `Q` as separate goals.

P ∨ Q

If you have $h : P$, then `Or.intro_left Q h` is a proof of $P \vee Q$. Usually there is no need to specify Q , and `Or.inl h` will be recognized as a proof of $P \vee Q$. Similarly, if you have $h : Q$, then `Or.intro_right P h` and `Or.inr h` are proofs of $P \vee Q$.

If you don't already have a proof of either P or Q , then the tactic `apply Or.inl` will set the goal to be P , and `apply Or.inr` will set the goal to be Q . Also, the tactic `or_left with h` will introduce the new given $h : \neg Q$ and set the goal to be P , and `or_right with h` will introduce the new given $h : \neg P$ and set the goal to be Q .

Proof by cases is sometimes useful for proving disjunctions. There is further information about [proof by cases](#) below.

P ↔ Q

If you have $h1 : P \rightarrow Q$ and $h2 : Q \rightarrow P$, then `Iff.intro h1 h2` is a proof of $P \leftrightarrow Q$. If you have $h : P \rightarrow Q$, then the tactic `apply Iff.intro h` will set the goal to be $Q \rightarrow P$, and if you have $h : Q \rightarrow P$, then `apply Iff.intro _ h` will set the goal to be $P \rightarrow Q$. If you don't already have a proof of either $P \rightarrow Q$ or $Q \rightarrow P$, then `apply Iff.intro` will set $P \rightarrow Q$ and $Q \rightarrow P$ as separate goals.

If P and Q are definitionally equal, then the tactic `rfl` will prove $P \leftrightarrow Q$.

∀ (x : U), P x

The tactic `fix x : U` will introduce the new variable $x : U$ and set the goal to be $P x$.

∃ (x : U), P x

If you have $a : U$ and $h : P a$, then `Exists.intro a h` is a proof of $\exists (x : U), P x$. If you have $a : U$ but you don't have a proof of $P a$, then the tactic `apply Exists.intro a` will set the goal to be $P a$.

∃! (x : U), P x

The tactic `exists_unique` will set $\exists (x : U), P x$ and $\forall (x_1 x_2 : U), P x_1 \rightarrow P x_2 \rightarrow x_1 = x_2$ as separate goals.

$\forall (n : \text{Nat}), P n$

The tactic `by_induc` will set $P 0$ and $\forall (n : \text{Nat}), P n \rightarrow P (n + 1)$ as separate goals. (If the goal is $\forall n \geq k, P n$, then the `by_induc` tactic will use k as the base case rather than 0 .) The tactic `by_strong_induc` will set the goal to be $\forall (n : \text{Nat}), (\forall n_1 < n, P n_1) \rightarrow P n$.

To Use a Given of the Form ...

$h : \neg P$

If you also have $h1 : P$, then $h h1$ is a proof of `False` and `absurd h1 h` will be recognized as a proof of any statement. If your goal is `False`, then `contradict h` will set the goal to be P . (See also the tactics for reexpressing negative statements under the [Reexpressing Statements](#) heading below. There is also further information about [proof by contradiction](#) below.)

$h : P \rightarrow Q$

If you also have $h1 : P$, then $h h1$ is a proof of Q . (See also the `contrapos` tactic under the [Reexpressing Statements](#) heading below.)

$h : P \wedge Q$

$h.\text{left}$ is a proof of P , and $h.\text{right}$ is a proof of Q .

$h : P \vee Q$

The tactic `by_cases` on h will initiate a proof by cases; in case 1, h will be changed to $h : P$, and in case 2 it will be changed to $h : Q$. If you want to preserve the original h , then `by_cases on h with h1` will introduce new givens $h1 : P$ in case 1 and $h1 : Q$ in case 2; `by_cases on h with h1, h2` will introduce new givens $h1 : P$ in case 1 and $h2 : Q$ in case 2. (There is further information about [proof by cases](#) below.)

If you have $h1 : \neg P$, then `disj_syll h h1` will change h to $h : Q$; if you want to preserve the original h , then `disj_syll h h1 with h2` will introduce the new given $h2 : Q$. Similarly, if you have $h1 : \neg Q$, then `disj_syll h h1` will change h to $h : P$, and `disj_syll h h1 with h2` will introduce the new given $h2 : P$.

`h : P ↔ Q`

`h.ltr` is a proof of $P \rightarrow Q$ and `h.rtl` is a proof of $Q \rightarrow P$. (See also the [rewrite tactic](#) under the [Reexpressing Statements](#) heading below.)

`h : ∀ (x : U), P x`

If you have `a : U`, then `h a` is a proof of $P a$.

`h : ∃ (x : U), P x`

The tactic `obtain (a : U) (h1 : P a)` from `h` will introduce both the new object `a : U` and the new given `h1 : P a`.

`h : ∃! (x : U), P x`

The tactic `obtain (a : U) (h1 : P a) (h2 : ∀ (x_1 x_2 : U), P x_1 → P x_2 → x_1 = x_2)` from `h` will introduce the new object `a : U` and new givens `h1 : P a` and `h2 : ∀ (x_1 x_2 : U), P x_1 → P x_2 → x_1 = x_2`.

Other Techniques That Can Be Used in Any Proof

Proof by Contradiction

The tactic `by_contra h` will introduce a new given `h` that is the negation of the goal, and set the goal to be `False`. Usually you will complete a proof by contradiction by proving contradictory statements `h1 : Q` and `h2 : ¬Q`. Once you have proven such contradictory statements, either `h2 h1` or `absurd h1 h2` can be used as a proof of `False`.

If you are doing a proof by contradiction (so your goal is `False`) and you plan to complete the proof by contradicting some hypothesis `h1`, then the tactic `contradict h1` will set the goal to be the negation of `h1`.

The tactic `contradict h1 with h` is shorthand for `by_contra h; contradict h1`.

Proof by Cases

If you have $h : P \vee Q$, then the tactic `by_cases` on h will break your proof into two cases based on h . In case 1, h will be changed to $h : P$, and in case 2, it will be changed to $h : Q$. The tactic `by_cases` on h with $h1$ will introduce new givens $h1 : P$ in case 1 and $h1 : Q$ in case 2. The tactic `by_cases` on h with $h1, h2$ will introduce new givens $h1 : P$ in case 1 and $h2 : Q$ in case 2.

Another way to initiate a proof by cases is with the tactic `by_cases h : P`, which will introduce the new givens $h : P$ in case 1 and $h : \neg P$ in case 2.

Working Backwards

If the expression $t _$ would prove the goal if the blank were replaced with a proof of some statement P , then the tactic `apply t` will set the goal to be P . More generally, if $t _ _ \dots _ _$ would prove the goal if the blanks were replaced with proofs of several statements, then `apply t` will set all of those statements to be separate goals. The expression t can also contain blanks within it that will generate additional goals. For example, if the goal is $P \wedge Q$ and you have $h : Q$, then `apply And.intro _ h` will set the goal to be P .

Making Assertions

If t is a term-mode proof of a statement P , then the tactic `have h : P := t` will introduce the new given $h : P$. If you want to give a tactic-mode proof to justify h , then you can write `have h : P := by` and then write a tactic-mode proof of P . The tactic-mode proof should be indented to set it off from the surrounding proof.

If t is a term-mode proof of a statement P and P is the goal, then the tactic `show P from t` will complete the proof. The tactic `exact t` will also complete the proof.

Reexpressing Statements

All of the tactics in this section apply by default to the goal; they can be applied to a given h by adding `at h`.

If t is a proof of $a = b$, then the tactic `rewrite [t]` will replace a anywhere it appears with b , and `rewrite [←t]` will replace b with a . Similarly, if t is a proof of $P \leftrightarrow Q$, then `rewrite [t]` will replace P with Q and `rewrite [←t]` will replace Q with P . If t contains blanks, Lean will try to fill them in to produce a proof that can be used as a rewriting rule; as with the `apply` tactic, blanks at the end of a proof can be left out. You can put a list of rewriting rules in the brackets (for example, `rewrite [t1, ←t2, t3]`), and the replacements will be performed one after another.

The tactic `rw` is the same as `rewrite`, except that after doing the rewriting, it tries to prove the goal by using the `rfl` tactic (see [Easy Proofs](#) below).

The tactic `define` will write out the definition of a statement. If you want to define just a subexpression of the statement, then you can write `define : [subexpression]`. For example, if the goal is $x \in A \cap B \rightarrow x \in C$, then `define : x \in A \cap B` will change the goal to $x \in A \wedge x \in B \rightarrow x \in C$.

The tactics `contrapos`, `demorgan`, `conditional`, `double_neg`, `bicond_neg`, and `quant_neg` rewrite statements by applying various logical laws; for details, see the tables about laws of [sentential](#) and [quantificational](#) logic. As with the `define` tactic, you can add `: [subexpression]` to apply the law to the subexpression. For example, if the goal is $P \wedge (Q \rightarrow \neg R)$, then the tactic `contrapos : Q \rightarrow \neg R` will change it to $P \wedge (R \rightarrow \neg Q)$.

If the goal is a negative statement, then the tactic `push_neg` will apply multiple laws about negations to “push” the negation as far as possible into the statement.

If e is an expression of type U , and you want to give it the name n in your proof, then the tactic `set n : U := e` will introduce the variable n to stand for e .

Easy Proofs

The tactic `rfl` proves statements of the form $a = a'$ or $a \leftrightarrow a'$, if a and a' are definitionally equal.

The tactic `decide` proves statements that can be verified to be true by means of a calculation.

The tactic `trivial` proves some “obviously” true statements.

There is some overlap among these tactics; for example, all three could be used to prove $2 + 2 = 4$.

Algebra

The tactic `ring` will try to prove a goal that is an equation by combining algebraic laws involving addition, subtraction, multiplication, and exponentiation with natural number exponents.

The tactic `linarith` will try to combine givens that are linear equations or inequalities to prove a goal that is a linear equation or equality, or to prove the goal `False` if the givens contradict each other.

If t is a proof of a statement asserting a relationship between two quantities, then the tactic `rel [t]` will try to prove a goal that can be obtained from that relationship by applying the same operation to both sides. The tactic will try to find a theorem in Lean’s library that says that the operation preserves the relationship, and if the theorem requires auxiliary positivity facts, it will try to prove those facts as well.

Calculational Proofs

If P , Q , R , and S are statements, then you can prove $P \leftrightarrow S$ by proving $P \leftrightarrow Q$, $Q \leftrightarrow R$, and $R \leftrightarrow S$. Such proofs can conveniently be written as calculational proofs. We often use calculational proofs with the `have` and `show` tactics. For example, we might write:

```
have h : P ↔ S :=
  calc P
    _ ↔ Q := [proof of P ↔ Q]
    _ ↔ R := [proof of Q ↔ R]
    _ ↔ S := [proof of R ↔ S]
```

The proofs of the individual lines can be either term-mode proofs or tactic-mode proofs (introduced with `by`). Calculational proofs can also be used to string together equations or inequalities to prove an equation or inequality.

Transitioning to Standard Lean

If you want to continue to use Lean to write mathematical proofs, you may want to learn more about Lean. A good place to start is the [Lean Community website](#). The resources there use “standard” Lean, which is somewhat different from the Lean in this book.

In a few cases we have used notation in this book that differs from standard Lean notation. For example, if h is a proof of $P \leftrightarrow Q$, then we have used $h.ltr$ and $h.rtl$ to denote proofs of the left-to-right and right-to-left directions of the biconditional. The standard Lean notation for these is $h.mp$ and $h.mpr$, respectively (“mp” and “mpr” stand for “modus ponens” and “modus ponens reverse”). As explained at the end of [Section 5.4](#), the notations `Pred U` and `Rel A B` denote the types $U \rightarrow \text{Prop}$ and $A \rightarrow B \rightarrow \text{Prop}$, respectively. Although `Rel` is standard notation (defined in Lean’s math library `mathlib`), `Pred` is not; the notation `BinRel A` is also not standard Lean. In place of `Pred U` you should use $U \rightarrow \text{Prop}$, and in place of `BinRel A` you should use `Rel A A`.

However, the biggest difference between the Lean in this book and standard Lean is that the tactics marked with an asterisk in the table above are not a part of standard Lean. If you want to learn to write proofs in standard Lean, you’ll need to learn replacements for those tactics. We discuss some such replacements below. Some of these replacements are built into Lean, and some are defined in `mathlib`.

- `assume`, `fix`

If you are proving $P \rightarrow Q$ and you want to begin by assuming $h : P$, in standard Lean you would begin your proof by writing `intro h`. You don't need to specify that h is an identifier for the assumption P ; Lean will figure that out on its own.

If you are proving $\forall (x : U), P x$ and you want to begin by introducing the variable x to stand for an arbitrary object of type U , in standard Lean you would begin your proof by writing `intro x`. Again, you don't need to specify the type of x , because Lean will figure it out.

Thus, the tactic `intro` does the job of both `assume` and `fix`. Furthermore, you can introduce multiple assumptions or objects with a single use of the `intro` tactic: `intro a b c` is equivalent to `intro a; intro b; intro c`.

- `bicond_neg`, `demorgan`, `double_neg`, `quant_neg`

We have mostly used these tactics to reexpress negative statements as more useful positive statements. The tactic `push_neg` can be used for this purpose.

- `by_cases on`

If you have $h : P \vee Q$, then you can break your proof into cases by using the tactic `rcases h with hP | hQ`. In case 1, $h : P \vee Q$ will be replaced by $hP : P$, and in case 2 it will be replaced by $hQ : Q$. In both cases, you have to prove the original goal.

- `by_induc`, `by_strong_induc`

We saw in Section 7.2 that if you are proving a statement of the form $\forall (l : List U), \dots$, then you can begin a proof by induction on the length of l by using the tactic `apply List.rec`. Similarly, if you are proving $\forall (n : Nat), \dots$, you can begin a proof by induction by using the tactic `apply Nat.recAux`. For strong induction, you can use `apply Nat.strongRec`.

There is also a tactic `induction` that you may want to learn about.

- `conditional`

The commands `#check @imp_iff_not_or` and `#check @not_imp` produce the results

```
@imp_iff_not_or : ∀ {a b : Prop}, a → b ↔ ¬a ∨ b
@not_imp : ∀ {a b : Prop}, ¬(a → b) ↔ a ∧ ¬b
```

Thus, `rewrite [imp_iff_not_or]` will convert a statement of the form $P \rightarrow Q$ into $\neg P \vee Q$, and `rewrite [←imp_iff_not_or]` will go in the other direction. Similarly, `rewrite [not_imp]` will convert a statement of the form $\neg(P \rightarrow Q)$ into $P \wedge \neg Q$, and `rewrite [←not_imp]` will go in the other direction.

- `contradict`

Suppose your goal is `False` (as it would be if you are doing a proof by contradiction), and you have `h : ¬P`. Recall that Lean treats `¬P` as meaning the same thing as `P → False`, and therefore `h _` will prove the goal, if the blank is filled in with a proof of `P`. It follows that `apply h` will set `P` as the goal. In other words, in this situation `apply h` has the same effect as `contradict h`.

You could also get the same effect with the tactic `suffices hP : P from h hP`. Think of this as meaning “it would suffice now to prove `P`, because if `hP` were a proof of `P`, then `h hP` would prove the goal.” Lean therefore sets `P` to be the goal.

Similarly, in a proof by contradiction, if you have `h : P`, then `suffices hnP : ¬P from hnP h` will set `¬P` as the goal.

Yet another possibility is `contrapose! h`. (This is a variant on the `contrapose!` tactic, discussed in the next section.)

- **contrapos**

If your goal is a conditional statement, then the tactics `contrapose` and `contrapose!` will replace the goal with its contrapositive (`contrapose!` also uses `push_neg` to try to simplify the negated statements that arise when forming a contrapositive). You may also find the theorem `not_imp_not` useful:

```
@not_imp_not : ∀ {a b : Prop}, ¬a → ¬b ↔ b → a
```

- **define**

The tactic `whnf` (which stands for “weak head normal form”) is similar to `define`, although it sometimes produces results that are a little confusing.

Another way to write out definitions is to prove a lemma stating the definition and then use that lemma as a rewriting rule in the `rewrite` tactic. See, for example, the use of the theorem `inv_def` in [Section 4.2](#).

- **disj_syll**

The following theorems can be useful:

```
@Or.resolve_left : ∀ {a b : Prop}, a ∨ b → ¬a → b
@Or.resolve_right : ∀ {a b : Prop}, a ∨ b → ¬b → a
@Or.neg_resolve_left : ∀ {a b : Prop}, ¬a ∨ b → a → b
@Or.neg_resolve_right : ∀ {a b : Prop}, a ∨ ¬b → b → a
```

For example, if you have `h1 : P ∨ Q` and `h2 : ¬P`, then `Or.resolve_left h1 h2` is a proof of `Q`.

- **exists_unique**

If your goal is $\exists! (x : U), P x$ and you think that a is the unique value of x that makes $P x$ true, then you can use the tactic `apply ExistsUnique.intro a`. This will leave you with two goals to prove, $P a$ and $\forall (y : U), P y \rightarrow y = a$.

- **obtain**

If you have $h : \exists (x : U), P x$, then the tactic `obtain <u, h1> := h` will introduce both $u : U$ and $h1 : P u$ into the tactic state. Note that u and $h1$ must be enclosed in angle brackets, $\langle \rangle$. To enter those brackets, type `\<` and `\>`.

If you have $h : \exists! (x : U), P x$, then `obtain <u, h1, h2> := h` will also introduce $u : U$ and $h1 : P u$ into the tactic state. In addition, it will introduce $h2$ as an identifier for a statement that is equivalent to $\forall (y : U), P y \rightarrow y = u$. (Unfortunately, the statement introduced is more complicated.)

You may also find the theorems `ExistsUnique.exists` and `ExistsUnique.unique` useful:

```
@ExistsUnique.exists : ∀ {α : Sort u_1} {p : α → Prop},
  (exists! (x : α), p x) → ∃ (x : α), p x
@ExistsUnique.unique : ∀ {α : Sort u_1} {p : α → Prop},
  (exists! (x : α), p x) → ∀ {y₁ y₂ : α}, p y₁ → p y₂ → y₁ = y₂
```

- **or_left, or_right**

If your goal is $P \vee Q$, then the tactics `or_left` and `or_right` let you assume that one of P and Q is false and prove the other. One way to do that in standard Lean is to use proof by cases. For example, to assume P is false and prove Q you might proceed as follows:

```
-- Goal is P ∨ Q
by_cases hP : P
· -- Case 1. hP : P
  exact Or.inl hP
  done
· -- Case 2. hP : ¬P
  apply Or.inr
  --We now have hP : ¬P, and goal is Q
  done
```

Another possibility is to use one of the theorems `Or.resolve_left`, `Or.resolve_right`, `Or.neg_resolve_left`, or `Or.neg_resolve_right` (described under `disj_syll` above) to convert the goal to an implication.

- **show**

There is a `show` tactic in standard Lean, but it works a little differently from the `show` tactic we have used in this book. When our goal was a statement P and we had an expression t that was a proof of P , we usually completed the proof by writing `show P from t`. In standard Lean you can complete the proof by writing `exact t`, as explained near the end of [Section 3.6](#).

Typing Symbols

Symbol	How To Type It
\neg	<code>\not</code> or <code>\n</code>
\wedge	<code>\and</code>
\vee	<code>\or</code> or <code>\v</code>
\rightarrow	<code>\to</code> or <code>\r</code> or <code>\imp</code>
\leftrightarrow	<code>\iff</code> or <code>\lr</code>
\forall	<code>\forallall</code> or <code>\all</code>
\exists	<code>\existsists</code> or <code>\ex</code>
$\{$	<code>\{\{</code>
$\}$	<code>\}\}</code>
$=$	<code>=</code>
\neq	<code>\ne</code>
\in	<code>\in</code>
\notin	<code>\notinin</code> or <code>\inn</code>
\subseteq	<code>\sub</code>
$\not\subseteq$	<code>\subn</code>
\cup	<code>\union</code> or <code>\cup</code>
\cap	<code>\inter</code> or <code>\cap</code>
\cup_0	<code>\U0</code>
\cap_0	<code>\I0</code>
\backslash	<code>\backslash</code>
Δ	<code>\symmdiff</code>
\emptyset	<code>\emptysetset</code>
\mathcal{P}	<code>\powerset</code>
\cdot	<code>\cdot</code>
\leftarrow	<code>\leftarrowarrow</code> or <code>\l</code>
\uparrow	<code>\uparrowarrow</code> or <code>\u</code>
\mathbb{N}	<code>\N</code>
\mathbb{Z}	<code>\Z</code>
\mathbb{Q}	<code>\Q</code>
\mathbb{R}	<code>\R</code>
\mathbb{C}	<code>\C</code>
\leq	<code>\le</code>

Typing Symbols

Symbol	How To Type It
\geq	<code>\ge</code>
$ $	<code>\ </code>
\times	<code>\times</code> or <code>\x</code>
\circ	<code>\comp</code> or <code>\circ</code>
\equiv	<code>\equiv</code>
\sim	<code>\sim</code> or <code>\~{}</code>
\simeq	<code>\simeq</code>
\langle	<code>\langle</code>
\rangle	<code>\rangle</code>