How To Prove It With Lean

Daniel J. Velleman

Table of contents

Preface

About This Book

This book is intended to accompany my book *How To Prove It* (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. To get the most out of this book, you will need to download and install Lean on your computer. We will explain how to do that below.

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 you will find a search box in the left margin. You can use that box 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.

Installing Lean

We will be using Visual Studio Code to run Lean, so you will need to install VS Code first. VS Code is free and can be downloaded here.

You will also need the Lean package that accompanies this book, which can be downloaded 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". Open the downloaded zip file to create a folder containing the HTPI Lean package. You can put this folder wherever you want on your computer.

Now open VS Code. You should see a window that looks something like this:

Click on the *Extensions* icon on the left side of the window, which is circled in red in the image above. That will bring up a list of available extensions:
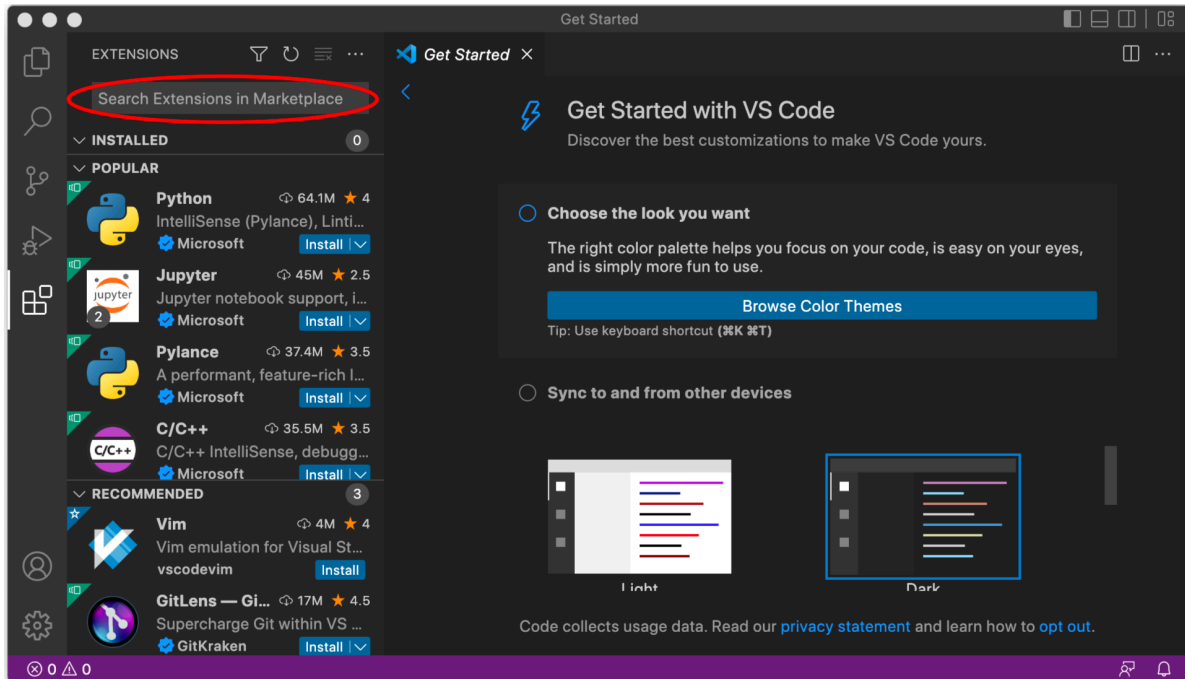
In the *Search Extensions in Marketplace* field, type "lean4". VS Code should find the Lean 4 extension and display it:

Click on "Install" to install the Lean 4 extension.

Next, in VS Code, select "Open Folder ..." from the File menu and open the folder containing the HTPI Lean package that you downloaded earlier. 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, circled in red below.)

Click on the file "Blank.lean" in the file list. You should see a warning that VS Code failed to start the 'lean' language server:

Click on the "Install Lean using Elan" button, and the Lean server should be installed. This may take a while, and there may be messages asking you to do things. If anything goes wrong, try quiting VS Code and restarting. Eventually your window should look like this:

If you don't see the Infoview pane on the right side of the window, click on the icon circled in red in the image above, and the Infoview pane should appear.

Your installation is now complete.

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. 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. 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. And we have the following operations on sets:

$A \cap B = \{x \mid x \in A \wedge x \in B\} =$ 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,

$A \triangle B = (A \setminus B) \cup (B \setminus A) =$ 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,

$$\mathscr{P}(A) = \{X \mid X \subseteq A\} = \text{ the } \textit{power set} \text{ of } A.$$

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

$$\bigcap \mathscr{F} = \{x \mid \forall A(A \in \mathscr{F} \to x \in A)\} = \text{ the } \textit{intersection} \text{ of the family } \mathscr{F},$$

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

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 $\exists x(P(x) \wedge \forall y(P(y) \to y = x))$.

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* (consult *HTPI* if you want to see how this proof was constructed):

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

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

And here is how we would write the proof in Lean:

```
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
```

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 hypotheses. 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 `¬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 $Q \to R$ and $\neg R$ to draw the inference $\neg Q$. This reasoning is justified by the contrapositive law, which says that $Q \to R$ is equivalent to its contrapositive, $\neg R \to \neg Q$. 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 `¬R → ¬Q`. Finally, in the last line of the proof, we combine the new `h4` with `h2` to infer `¬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 installed Lean on your computer, go back and follow the instructions for installing it now. Then in VS Code, open the folder containing the HTPI Lean Package that you downloaded and click on the file Blank.lean. The file starts with the line `import 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:

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

This theorem and proof are so short 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 message in the Lean Infoview pane explaining what the error is. (If you don't see the Infoview pane, choose "Command Palette …" in the "View" menu, and then type "Lean" in the text box that appears. You will see a list of commands that start with "Lean". Click on "Lean 4: Infoview: Toggle" 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 squiggle and error message go away.

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 `→` 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 `→`. 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 `→`.

```
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 → 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 → 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 → 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 → 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 ¬, type `\not`, followed again by either space or tab. Alternatively, if you type `Not P`, Lean will interpret it as meaning `¬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 `:=`. Although it is not necessary, 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.

One of the advantages of tactic mode is that Lean displays, in the Lean Infoview pane, information about the status of the proof as your 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:

Lean File   Tactic State in Infoview

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

  done
```

```
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 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   Tactic State in Infoview

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

```
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 → ¬R`. To prove the new goal, we will assume `P` and prove `¬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 `¬R`:

```
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
```

```
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 → 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`:

```
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
```

```
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 `¬R` from `h2 : Q → ¬R` and `h4 : Q`, using the term-mode proof `h2 h4`. Type `show ¬R from h2 h4` to complete the proof. You'll notice two changes in the display: the red squiggle will disappear from the word `done`, and the tactic state will say "Goals accomplished":

```
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
```

```
Goals accomplished 🎉
```

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.

We have now seen 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 → 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 → and ¬, 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
¬	\not or \n	Not
∧	\and	/\
∨	\or or \v	\/
→	\to or \r or \imp	->
↔	\iff or \lr	<->

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 important of these conventions is that `P → Q → R` is interpreted as `P → (Q → R)`, not `(P → Q) → R`. The reason for this is simply that statements of the form `P → (Q → R)` come up much more often in proofs than statements of the form `(P → Q) → R`. (Lean also follows this "grouping-to-the-right" convention for ∧ and ∨, 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 → Q	is changed to	¬Q → ¬P
De Morgan's Laws	`demorgan`	¬(P ∧ Q)	is changed to	¬P ∨ ¬Q
		¬(P ∨ Q)	is changed to	¬P ∧ ¬Q
		P ∧ Q	is changed to	¬(¬P ∨ ¬Q)
		P ∨ Q	is changed to	¬(¬P ∧ ¬Q)
Conditional Laws	`conditional`	P → Q	is changed to	¬P ∨ Q
		¬(P → Q)	is changed to	P ∧ ¬Q
		P ∨ Q	is changed to	¬P → Q
		P ∧ Q	is changed to	¬(P → ¬Q)
Double Negation Law	`double_neg`	¬¬P	is changed to	P
Biconditional Negation Law	`bicond_neg`	¬(P ↔ Q)	is changed to	¬P ↔ Q
		P ↔ Q	is changed to	¬(¬P ↔ 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

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

Lean File   Tactic State in Infoview

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

  done
```

```
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 ∈ B and a ∉ B \ C, and the conclusion, a ∈ C.

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   Tactic State in Infoview

```
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
```

```
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 ∈ B ∨ a ∈ C and then a conditional law to change it to a ∈ B → a ∈ C:

Lean File   Tactic State in Infoview

```
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
```

```
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   Tactic State in Infoview

```
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
```

Goals accomplished 🎉

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 → Q`:

To prove a goal of the form `P → 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 automaticall 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   Tactic State After Using Strategy

```
    ⋮                                            ⋮
  ⊢ P → Q                                      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 `¬Q → ¬P` and then use the tactic `assume h : ¬Q`. The net effect of these two tactics is:

Tactic State Before Using Strategy   Tactic State After Using Strategy

```
    ⋮                                            ⋮
  ⊢ P → Q                                      h : ¬Q
                                               ⊢ ¬P
```

Section 3.2 gives two strategies for using givens of the form `P → Q`, with the second once again being a variation on the first based on the contrapositive law:

To use a given of the form `P → 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 `¬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 → 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 → Q` and `h2 : ¬Q`, then the `contrapos at h1` tactic will change `h1` to `h1 : ¬Q → ¬P`, and then `h1 h2` will be a proof of `¬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:

To prove a goal of the form `¬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 `¬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
⋮	⋮
⊢ ¬P	`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 : ¬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 : ¬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 `¬Q` as meaning the same

thing as `Q → False`. This makes sense, because these statements are logically equivalent, as shown by the following truth table:

Q	¬Q	(Q	→	False)
F	T	F	T	F
T	F	T	F	F

Thinking of `h2 : ¬Q` as meaning `h2 : Q → 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 : ¬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 : ¬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 : ¬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 : ¬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. (We won't bother to include the `done` line in the displays below.)

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
```

Tactic State in Infoview

```
P Q R : Prop
h : P → Q → R
h2 : ¬R
h3 : P
⊢ ¬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
```

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 → 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
```

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 : ¬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
```

Tactic State in Infoview

```
Goals accomplished 🎉
```

Finally, we have two strategies for using a given that is a negative statement:

To use a given of the form ¬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 `¬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 : ¬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 `¬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 → ¬Q` and `h2 : P`, then `h1 h2` is a proof of `¬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
```

Tactic State in Infoview

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

Since we are now doing a proof by contradiction and the given `h2 : ¬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
```

Tactic State in Infoview

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

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 → 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
```

Tactic State in Infoview

```
Goals accomplished 🎉
```

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
```

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 ∈ C
```

The goal is `a ∈ C`, and the only given that even mentions `C` is `h3 : a ∈ B → a ∈ C`. If only we could prove `a ∈ 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 ∈ 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 ∈ C`, what do we have to put in the blank? Lean is able to figure out that the answer is: a proof of `a ∈ B`. So it sets the goal to be `a ∈ B`, since a proof of that goal, when inserted into the blank in `h3 _`, would prove the original goal `a ∈ C`:

Lean File	Tactic State in Infoview

```
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 _
```

```
U : Type
A B C: Set U
a : U
h1 : a ∈ A
h2 : ¬a ∈ A \ B
h3 : a ∈ B → a ∈ C
⊢ a ∈ B
```

It may not be clear what to do next, but the given `h2` is a negative statement, so perhaps reexpressing it will help. Writing out the definition of set difference, `h2` means ¬(a ∈ A ∧ a ∉ B), and then one of De Morgan's laws and a conditional law allow us to rewrite it first as a ∉ A ∨ a ∈ B and then as a ∈ A → a ∈ B. Of course, we have tactics to accomplish all of these reexpressions:

Lean File	Tactic State in Infoview

```
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
```

```
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	Tactic State in Infoview

```
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
```

```
Goals accomplished 🎉
```

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

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.

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 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 : 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 `Pred U`, then to say that `P` holds for all objects of type `U` we would write `∀ (x : U), P x`. Similarly, `∃ (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, $\forall x\, P(x) \land Q(x)$ is interpreted as $(\forall x\, P(x)) \land Q(x)$; if you want the quantifier $\forall x$ to apply to the entire statement $P(x) \land Q(x)$ you must use parentheses and write $\forall x(P(x) \land Q(x))$. The convention in Lean is exactly the opposite: a quantifier applies to as much as possible. Thus, Lean will interpret `∀ (x : U), P x ∧ Q x` as meaning `∀ (x : U), (P x ∧ Q x)`. If you want the quantifier to apply to only `P x`, then you must use parentheses and write `(∀ (x : U), P x) ∧ 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 `∀ (x : U), P x` is to use the following strategy:

To prove a goal of the form `∀ (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	Tactic State After Using Strategy

```
    ⋮
 ⊢ ∀ (x : U), P x
```

```
    ⋮
 x : U
 ⊢ P x
```

To use a given of the form `∀ (x : U), P x`, we usually apply a rule of inference called *universal instantiation*, which is described by the following proof strategy:

To use a given of the form `∀ (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 : ∀ (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.

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 theorem, you can enter the symbol `∀` by typing `\forall` or `\all`, and you can enter `∃` by typing `\exists` or `\ex`.

Lean File	Tactic State in Infoview

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

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

(In the tactic state, why doesn't Lean show the type of the variable `x` after the existential quantifier in the goal? I don't know. Sometimes you can leave out the type of a quantified variable and Lean is able to figure it out on its own. But sometimes Lean is unable to figure out the type if it is not supplied, and you will get an error message if you leave it out. To avoid confusion, we will always include the type of the quantified variable when we enter a quantified statement into Lean, but you will notice that Lean generally leaves out the type when it displays existentially quantified statements in the tactic state.)

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		
¬∀ (x : U), P x	is changed to	∃ (x : U), ¬P x
¬∃ (x : U), P x	is changed to	∀ (x : U), ¬P x
∀ (x : U), P x	is changed to	¬∃ (x : U), ¬P x
∃ (x : U), P x	is changed to	¬∀ (x : U), ¬P x

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

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
```

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 ∀, 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
```

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
```

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 → ¬P y, then we can apply modus ponens to the rewritten h3 and h4 to reach the goal:

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
```

Tactic State in Infoview

```
Goals accomplished 🎉
```

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
∉	\notin or \inn
⊆	\sub
⊄	\subn
∪	\union or \cup
∩	\inter or \cap
\	\\
△	\bigtriangleup
∅	\emptyset

Symbol	How To Type It
$\mathscr{P}$	\powerset

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

Lean File   Tactic State in Infoview

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

```
U : Type
A B C : Set U
h1 : A ⊆ B ∪ C
h2 : ∀ (x : U),
  x ∈ A → ¬x ∈ B
⊢ A ⊆ C
```

We begin by using the `define` tactic to write out the definition of the goal.

Lean File   Tactic State in Infoview

```
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
```

```
U : Type
A B C: Set U
h1 : A ⊆ B ∪ C
h2 : ∀ (x : U),
  x ∈ A → ¬x ∈ B
⊢ ∀ {a : U},
  a ∈ A → a ∈ C
```

Notice that Lean's definition of the goal starts with ∀ {a : U}, not ∀ (a : U). Why did Lean use curly 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 ∈ A.

Lean File   Tactic State in Infoview

```
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
```

```
U : Type
A B C : Set U
h1 : A ⊆ B ∪ C
h2 : ∀ (x : U),
  x ∈ A → ¬x ∈ B
y : U
h3 : y ∈ A
⊢ y ∈ C
```

Now we can combine `h2` and `h3` to conclude that `¬y ∈ B`. Since we have `y : U`, by universal instantiation, `h2 y` is a proof of `y ∈ A → ¬y ∈ B`, and therefore by modus ponens, `h2 y h3` is a proof of `¬y ∈ 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
```

Tactic State in Infoview

```
U : Type
A B C : Set U
h1 : A ⊆ B ∪ C
h2 : ∀ (x : U),
  x ∈ A → ¬x ∈ B
y : U
h3 : y ∈ A
h4 : ¬y ∈ B
⊢ y ∈ 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 ∈ U → a ∈ B ∪ C
```

Tactic State in Infoview

```
U : Type
A B C : Set U
h1 : ∀ {a : U},
  a ∈ A → a ∈ B ∪ C
h2 : ∀ (x : U),
  x ∈ A → ¬x ∈ B
y : U
h3 : y ∈ A
h4 : ¬y ∈ B
⊢ y ∈ C
```

Once again, Lean has used curly 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 curly 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`.

34

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 implicit. We will see many more examples of implicit arguments later in this book.

Lean File   Tactic State in Infoview

```
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 ∈ U → a ∈ B ∪ C
  have h5 : y ∈ B ∪ C := h1 h3
```

```
U : Type
A B C : Set U
h1 : ∀ {a : U},
  a ∈ A → a ∈ B ∪ C
h2 : ∀ (x : U),
  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	Tactic State in Infoview

```
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 ∈ U → a ∈ B ∪ C
  have h5 : y ∈ B ∪ C := h1 h3
  define at h5  --h5 : y ∈ B ∨ y ∈ C
```

```
U : Type
A B C : Set U
h1 : ∀ {a : U},
  a ∈ A → a ∈ B ∪ C
h2 : ∀ (x : U),
  x ∈ A → ¬x ∈ B
y : U
h3 : y ∈ A
h4 : ¬y ∈ B
h5 : y ∈ B ∨ y ∈ C
⊢ y ∈ C
```

A conditional law will convert `h5` to `¬y ∈ B → y ∈ C`, and then modus ponens with `h4` will complete the proof.

Lean File	Tactic State in Infoview

```
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 ∈ U → 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
```

```
Goals accomplished 🎉
```

Next we turn to strategies for working with existential quantifiers.

To prove a goal of the form `∃ (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 `∃ (x : U), P x`. Indeed, in this situation the expression `Exists.intro a h` is a Lean term-mode proof of `∃ (x : U), P x`. The name `Exists.intro` indicates that this is a rule for introducing an existential quantifier.

As suggested by the strategy above, we will often want to use this rule in situations in which our goal is `∃ (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
```

Our strategy for using an existential given is a rule that is called *existential instantiation* in *HTPI*:

To use a given of the form `∃ (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.

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

Often, if your goal is an existential statement `∃ (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
```

Tactic State in Infoview

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

The goal is the existential statement `∃ (x : U), ¬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

`¬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.

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
```

Tactic State in Infoview

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

Tactic State in Infoview

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

Tactic State in Infoview

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

Tactic State in Infoview

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

Lean File	Tactic State in Infoview

```
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 _
```

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

```
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
```

```
U : Type
P Q : Pred U
h1 : ∀ (x : U), ∃ y,
  P x → ¬Q y
h2 : ∃ x, ∀ (y : U),
  P x → Q y
a : U
h3 : ∀ (y : U), P a → Q y
h4 : ∃ y, 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 `¬Q b` and `h6 h7` is a proof of `Q b`, so `h5 h7 (h6 h7)` is a proof of `False`.

```
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)
```

Goals accomplished 🎉

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

Tactic State in Infoview

```
theorem Example_3_3_5 (U : Type) (B : Set U)
    (F : Set (Set U)) : ∪₀F ⊆ B → F ⊆ 𝒫 B := by
```

```
U : Type
B : Set U
F : Set (Set U)
⊢ ∪₀F ⊆ B → F ⊆ 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

Tactic State in Infoview

```
theorem Example_3_3_5 (U : Type) (B : Set U)
    (F : Set (Set U)) : ∪₀F ⊆ B → F ⊆ 𝒫 B := by
  assume h1 : ∪₀F ⊆ B
  define
```

```
U : Type
B : Set U
F : Set (Set U)
h1 : ∪₀F ⊆ B
⊢ ∀ {a : Set U},
    a ∈ F → a ∈ 𝒫 B
```

Based on the form of the goal, we introduce an arbitrary object x of type Set U and assume x ∈ F. The new goal will be x ∈ 𝒫 B. The define tactic works out that this means x ⊆ B, which can be further expanded to ∀ {a : U}, a ∈ x → a ∈ B.

Lean File

```
theorem Example_3_3_5 (U : Type) (B : Set U)
    (F : Set (Set U)) : U₀F ⊆ B → F ⊆ 𝒫 B := by
  assume h1 : U₀F ⊆ B
  define
  fix x : Set U
  assume h2 : x ∈ F
  define
```

Tactic State in Infoview

```
U : Type
B : Set U
F : Set (Set U)
h1 : U₀F ⊆ 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 ∈ x.

Lean File

```
theorem Example_3_3_5 (U : Type) (B : Set U)
    (F : Set (Set U)) : U₀F ⊆ B → F ⊆ 𝒫 B := by
  assume h1 : U₀F ⊆ B
  define
  fix x : Set U
  assume h2 : x ∈ F
  define
  fix y : U
  assume h3 : y ∈ x
```

Tactic State in Infoview

```
U : Type
B : Set U
F : Set (Set U)
h1 : U₀F ⊆ 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)) : U₀F ⊆ B → F ⊆ 𝒫 B := by
  assume h1 : U₀F ⊆ B
  define
  fix x : Set U
  assume h2 : x ∈ F
  define
  fix y : U
  assume h3 : y ∈ x
  define at h1
```

Tactic State in Infoview

```
U : Type
B : Set U
F : Set (Set U)
h1 : ∀ {a : U},
    a ∈ U₀F → a ∈ B
x : Set U
h2 : x ∈ F
y : U
h3 : y ∈ x
⊢ y ∈ B
```

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 ∈ U₀F. So we use the apply h1 _ tactic.

Lean File	Tactic State in Infoview

```
theorem Example_3_3_5 (U : Type) (B : Set U)
    (F : Set (Set U)) : U₀F ⊆ B → F ⊆ 𝒫 B := by
  assume h1 : U₀F ⊆ B
  define
  fix x : Set U
  assume h2 : x ∈ F
  define
  fix y : U
  assume h3 : y ∈ x
  define at h1
  apply h1 _
```

U : Type
B : Set U
F : Set (Set U)
$h1$: ∀ {a : U},
  a ∈ U₀F → a ∈ B
x : Set U
$h2$: x ∈ F
y : U
$h3$: y ∈ x
⊢ y ∈ U₀F

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

Lean File	Tactic State in Infoview

```
theorem Example_3_3_5 (U : Type) (B : Set U)
    (F : Set (Set U)) : U₀F ⊆ B → F ⊆ 𝒫 B := by
  assume h1 : U₀F ⊆ B
  define
  fix x : Set U
  assume h2 : x ∈ F
  define
  fix y : U
  assume h3 : y ∈ x
  define at h1
  apply h1 _
  define
```

U : Type
B : Set U
F : Set (Set U)
$h1$: ∀ {a : U},
  a ∈ U₀F → a ∈ B
x : Set U
$h2$: x ∈ F
y : U
$h3$: y ∈ x
⊢ ∃ a, a ∈ F ∧ y ∈ a

Our goal is now an existential statement, so we look for a value of `a` that will make the statement `a ∈ F ∧ y ∈ a` 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 _`.

```
theorem Example_3_3_5 (U : Type) (B : Set U)
    (F : Set (Set U)) : U₀F ⊆ B → F ⊆ 𝒫 B := by
  assume h1 : U₀F ⊆ 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 _
```

```
U : Type
B : Set U
F : Set (Set U)
h1 : ∀ {a : U},
  a ∈ U₀F → 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 ∧, 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.

```
theorem Example_3_3_5 (U : Type) (B : Set U)
    (F : Set (Set U)) : U₀F ⊆ B → F ⊆ 𝒫 B := by
  assume h1 : U₀F ⊆ 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
```

```
Goals accomplished 🎉
```

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*:

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

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 $x \in \mathscr{P}(B)$. But x was an arbitrary element of $\mathcal{F}$, so this shows that $\mathcal{F} \subseteq \mathscr{P}(B)$, as required. $\qquad\square$

3.4. Proofs Involving Conjunctions and Biconditionals

The strategies in *HTPI* for working with conjunctions are very simple.

To prove a goal of the form `P ∧ 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 ∧ Q`. It follows that if your goal is `P ∧ 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.

To use a given of the form `P ∧ Q`:

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

If you have a given `h : P ∧ 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
```

Tactic State in Infoview

```
U : Type
A B C D : Set U
h1 : A ⊆ B
h2 : ¬∃ c, c ∈ C ∩ D
⊢ A ∩ C ⊆ B \ D
```

The `define` tactic will rewrite the goal as ∀ {a : U}, a ∈ A ∩ C → a ∈ B \ D, and then we can introduce an arbitrary x : U and assume x ∈ A ∩ C.

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
```

Tactic State in Infoview

```
U : Type
A B C D : Set U
h1 : A ⊆ B
h2 : ¬∃ c, 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
```

Tactic State in Infoview

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

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

```
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
```

```
case left
U : Type
A B C D : Set U
h1 : A ⊆ B
h2 : ¬∃ c, 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, 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 proving the left side of the conjunction and the second is proving the right. The given information in both cases is the same, but in the first case the goal is x ∈ B, and in the second it is ¬x ∈ D. As we 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. To make the proof more readable, we will add comments indicating which steps give the proof of the first goal and which prove the second. (If, for some reason, you prefer to prove the second goal first, the tactic swap will switch the order of the two goals.)

The 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: 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.

Lean File	Tactic State in Infoview

```
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
```

```
case right
U : Type
A B C D : Set U
h1 : A ⊆ B
h2 : ¬∃ c, c ∈ C ∩ D
x : U
h3 : x ∈ A ∧ x ∈ C
⊢ ¬x ∈ D
```

Note that Lean has recognized that the first goal has been achieved, so it has moved on to the second goal. This 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	Tactic State in Infoview

```
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
  -- Proof that ¬x ∈ D:
  contradict h2 with h4
```

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

Lean File	Tactic State in Infoview

```
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
  -- Proof that ¬x ∈ D:
  contradict h2 with h4
  apply Exists.intro x
```

```
case right
U : Type
A B C D : Set U
h1 : A ⊆ B
h2 : ¬∃ c, 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 ∈ C ∧ x ∈ 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	Tactic State in Infoview

```
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
  -- Proof that ¬x ∈ D:
  contradict h2 with h4
  apply Exists.intro x
  show x ∈ C ∩ D from And.intro h3.right h4
```

```
Goals accomplished 🎉
```

Since P ↔ Q is shorthand for (P → Q) ∧ (Q → P), the strategies given above for conjunctions lead immediately to the following strategies for biconditionals:

To prove a goal of the form `P ↔ Q`:

Prove `P → Q` and `Q → P` separately.

To use a given of the form `P ↔ Q`:

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

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

Let's try these strategies out in an example.

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
```

Tactic State in Infoview

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

Tactic State in Infoview

```
case mp
U : Type
P Q : Pred U
h1 : ∀ (x : U), P x ↔ Q x
⊢ (∃ x, P x) → ∃ x, Q x
case mpr
U : Type
P Q : Pred U
h1 : ∀ (x : U), P x ↔ Q x
⊢ (∃ x, Q x) → ∃ x, 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".) As in *HTPI*, we'll label the proofs of the two goals with (→) and (←), representing the two directions of the biconditional symbol ↔. (You can type ← in VS Code by typing \l, short for "left".) The first goal is a

50

conditional statement, so we assume the antecedent. In the displays below, we'll just show the tactic state for the first goal, since that's all that's relevant for figuring out our next steps.

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
```

Tactic State in Infoview

```
case mp
U : Type
P Q : Pred U
h1 : ∀ (x : U), P x ↔ Q x
h2 : ∃ x, P x
⊢ ∃ x, Q x
```

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
```

Tactic State in Infoview

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

Tactic State in Infoview

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

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
  apply Exists.intro u
```

Tactic State in Infoview

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

To complete the proof, we use the left-to-right direction of `h4`. We have `h4.ltr : P u → Q u` and `h3 : P u`, so by modus ponens, `h4.ltr h3` proves the goal `Q u`.

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
  apply Exists.intro u
  show Q u from h4.ltr h3
```

Tactic State in Infoview

```
case mpr
U : Type
P Q : Pred U
h1 : ∀ (x : U), P x ↔ Q x
⊢ (∃ x, Q x) → ∃ x, P x
```

This completes the (→) half of the proof; the tactic state now lists only the goal for the (←) half. The second half of the proof is similar to the first. We begin by assuming `h2 : ∃ (x : U), Q x`, and then we use that assumption to obtain `u : U` and `h3 : Q u`.

| Lean File | Tactic State in Infoview |

```
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
  apply Exists.intro u
  show Q u from h4.ltr h3
  -- (←)
  assume h2 : ∃ (x : U), Q x
  obtain (u : U) (h3 : Q u) from h2
```

```
case mpr
U : Type
P Q : Pred U
h1 : ∀ (x : U), P x ↔ Q x
h2 : ∃ x, Q x
u : U
h3 : Q u
⊢ ∃ x, 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) :
    (∃ (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
  apply Exists.intro u
  show Q u from h4.ltr h3
  -- (←)
  assume h2 : ∃ (x : U), Q x
  obtain (u : U) (h3 : Q u) from h2
  show ∃ (x : U), P x from Exists.intro u ((h1 u).rtl h3)
```

To understand the last step, start with the fact that h1 u is a proof of P u ↔ Q u. Therefore (h1 u).rtl is a proof of Q u → P u, so by modus ponens, (h1 u).rtl h3 is a proof of P u. It follows that Exists.intro u ((h1 u).rtl h3) is a proof of ∃ (x : U), P x, which was the goal.

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*.

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 $A \cap (B \setminus C) = (A \cap B) \setminus C$. □

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 _
```

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 `∀ (x : U), x ∈ A ∩ (B \ C) ↔ x ∈ (A ∩ B) \ 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.

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-hand side of each biconditional after the first as standing for the right-hand side of the previous biconditional.

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`. So `Iff.refl` is the name of the theorem `∀ (a : Prop), a ↔ a`. (This theorem says that "iff" has a property called *reflexivity*; we'll discuss reflexivity in Chapter 4.) Thus, by universal instantiation, for any proposition a, `Iff.refl a` is 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 ∈ A ∩ (B \ C) ↔ x ∈ A ∧ (x ∈ B ∧ x ∉ 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 line of the calculational proof becomes x ∈ A ∧ (x ∈ B ∧ x ∉ C) ↔ x ∈ A ∧ (x ∈ B ∧ x ∉ 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 a response from Lean that looks something like this:

```
and_assoc : (?m.4075 ∧ ?m.4076) ∧ ?m.4077 ↔ ?m.4075 ∧ ?m.4076 ∧ ?m.4077
```

The explanation for this cryptic response is that `and_assoc` has implicit arguments, and they are not displayed by default. To see the implicit arguments, you must type `#check @and_assoc`. Lean's response is:

```
@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 Lean groups the logical connectives to the right, which means that it interprets a ∧ b ∧ c as a ∧ (b ∧ 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 ∧ b) ∧ c ↔ a ∧ (b ∧ 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 ∈ A ∧ (x ∈ B ∧

$x \notin C) \leftrightarrow (x \in A \wedge x \in B) \wedge x \notin C$, 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 biconditional 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 ↔ Q`, then `h.symm` is a proof of `Q ↔ 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 `(∀ (x : α), x ∈ a ↔ x ∈ b) → 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`.

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.

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.

To use a given of the form `P ∨ 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 ∨ 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 ∨ 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 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 two 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 h` as a proof of `P ∨ Q`, and if you have `h : Q`, then `Or.intro_right h` is a proof of `P ∨ Q`. There are shortened forms `Or.inl` and `Or.inr` for `Or.intro_left` and `Or.intro_right`, respectively.

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.

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
```

Tactic State in Infoview

```
U : Type
A B C : Set U
⊢ A \ (B \ C) ⊆ A \ B ∪ C
```

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, then there is no need to use it. After introducing x as an arbitrary element of A \ (B \ 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
```

Tactic State in Infoview

```
U : Type
A B C : Set U
x : U
h1 : x ∈ A ∧ ¬x ∈ B \ C
⊢ x ∈ A \ B ∨ x ∈ 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 ∈ B \ C is ¬(x ∈ B ∧ x ∉ C), which, by one of the De Morgan laws, is equivalent to x ∉ B ∨ x ∈ 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
          --Now h2 : ¬x ∈ B ∨ x ∈ C
```

Tactic State in Infoview

```
U : Type
A B C : Set U
x : U
h1 : x ∈ A ∧ ¬x ∈ B \ C
h2 : ¬x ∈ B ∨ x ∈ C
⊢ x ∈ A \ B ∨ x ∈ 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
          --Now h2 : ¬x ∈ B ∨ x ∈ C
  by_cases on h2
```

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
```

Looking at the givens h1 and h2 in both cases, it is not hard to see that we should be able to prove x ∈ A \ B in case 1 and x ∈ 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 ∈ A \ B, and in case 2 we can use Or.inr _, filling in the blank with a proof of x ∈ 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
          --Now h2 : ¬x ∈ B ∨ x ∈ C
  by_cases on h2
  -- Case 1
  apply Or.inl
```

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
```

Notice that the tactic apply Or.inl has changed the goal for case 1 to the left half of the original goal, x ∈ A \ B. Since this means x ∈ A ∧ x ∉ B, we can complete case 1 by combining h1.left with h2.

Lean File	Tactic State in Infoview

```
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
            --Now h2 : ¬x ∈ B ∨ x ∈ C
  by_cases on h2
  -- Case 1
  apply Or.inl
  show x ∈ A \ B from And.intro h1.left h2
```

```
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	Tactic State in Infoview

```
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
            --Now h2 : ¬x ∈ B ∨ x ∈ C
  by_cases on h2
  -- Case 1
  apply Or.inl
  show x ∈ A \ B from And.intro h1.left h2
  -- Case 2
  apply Or.inr
  show x ∈ C from h2
```

```
Goals accomplished 🎉
```

There is a second strategy that is often useful to prove a goal of the form `P ∨ Q`. It is motivated by the fact that `P ∨ Q` is equivalent to both `¬P → Q` and `¬Q → P`.

To prove a goal of the form `P ∨ Q`:

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

If your goal is `P ∨ Q`, then the Lean tactic `or_left with h` will add the new given `h : ¬Q` to the tactic state and set the goal to be `P`, and `or_right with h` will add `h : ¬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	Tactic State After Using Strategy
⋮ ⊢ P ∨ Q	⋮ 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 extra given might be useful, you are better off using `or_left` or `or_right`. Here's an example illustrating this.

Lean File	Tactic State in Infoview

```example (U : Type) (A B C : Set U)```
 ```   (h1 : A \ B ⊆ C) : A ⊆ B ∪ C := by```
	`U : Type`
	 `A B C : Set U`
	 `h1 : A \ B ⊆ C`
	 `⊢ A ⊆ B ∪ 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	Tactic State in Infoview

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

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

Lean File	Tactic State in Infoview

```
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
```

```
U : Type
A B C : Set U
h1 : A \ B ⊆ C
x : U
h2 : x ∈ A
h3 : ¬x ∈ B
⊢ x ∈ C
```

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

Lean File	Tactic State in Infoview

```
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)
```

```
Goals accomplished 🎉
```

The fact that `P ∨ Q` is equivalent to both `¬P → Q` and `¬Q → P` also suggests another strategy for using a given that is a disjunction.

To use a given of the form `P ∨ Q`:

If you are also given `¬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 `¬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 ∨ Q` and `h2 : ¬P`, then the tactic `disj_syll h1 h2` will change `h1` to `h1 : Q`; if instead you have `h2 : ¬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
```

Tactic State in Infoview

```
U : Type
A B C : Set U
h1 : A ⊆ B ∪ C
h2 : ¬∃ x, x ∈ A ∩ B
⊢ A ⊆ 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
```

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
```

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 ∈ B`, and then combining that conclusion with `h4` we can infer `a ∈ 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   --Now h5 : ¬a ∈ B
  disj_syll h4 h5   --Now h4 : a ∈ C
  show a ∈ C from h4
```

Tactic State in Infoview

```
Goals accomplished 🎉
```

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
```

Tactic State in Infoview

```
U : Type
A B C : Set U
h1 : A ⊆ B ∪ C
h2 : ¬∃ x, 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	Tactic State in Infoview

```
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  --Now h4 : a ∈ C
  show a ∈ C from h4
```

Goals accomplished 🎉

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 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	Tactic State in Infoview

```
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   --Now h4 : a ∈ C
  show a ∈ C from h4
```

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

Note that `h5 : a ∉ B` is not a given in the tactic state, because we have not yet justified it; in fact, `¬a ∈ 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	Tactic State in Infoview

```
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   --Now h4 : a ∈ C
  show a ∈ C from h4
```

```
U : Type
A B C : Set U
h1 : A ⊆ B ∪ C
h2 : ¬∃ x, x ∈ A ∩ B
a : U
h3 : a ∈ A
h4 : a ∈ B ∨ a ∈ C
h6 : a ∈ B
⊢ ∃ x, 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 ∈ A ∩ B`. Since this means `a ∈ A ∧ a ∈ B`, we can prove it with `And.intro h3 h6`. Thus, we can complete the proof in one more step:

Lean File	Tactic State in Infoview

```
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   --Now h4 : a ∈ C
  show a ∈ C from h4
```

Tactic State in Infoview:

```
Goals accomplished 🎉
```

The red squiggle has disappeared from the word `done`, 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"!

3.6. Existence and Uniqueness Proofs

Recall that `∃! (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 `∃ (x : U), P x ∧ ∀ (x_1 : U), P x_1 → 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 `∃! (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 ∧ B → C` is equivalent to `A → B → 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 ∧ B`

→ C as A → B → C. That's what we have done here, as you will see if you compare the strategy stated below to the one in *HTPI*.

To prove a goal of the form ∃! (x : U), P x:

> Prove ∃ (x : U), P x and ∀ (x_1 x_2 : U), P x_1 → P x_2 → 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*.

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*:

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 $C = D$. □

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, ∅ ∪ B = B, and it brings up a subtle issue: in Lean, the symbol ∅ 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 ∅ denote? The answer will be different in different contexts. Lean can sometimes 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 (∅ : Set U) rather than ∅.

With that preparation, we are ready to prove our first preliminary theorem. To avoid problems with ambiguity, we'll tell Lean explicitly which empty set we have in mind in this theorem. Since the goal is an equation between sets, our first step is to use the tactic apply Set.ext.

Lean File	Tactic State in Infoview

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

```
case h
U : Type
B : Set U
⊢ ∀ (x : U),
    x ∈ ∅ ∪ B ↔ x ∈ 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	Tactic State in Infoview

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

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

Of course, our next step is to assume x ∈ (∅ : Set U) ∪ B. To help us see how to move forward, we also write out the definition of this assumption.

Lean File	Tactic State in Infoview

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

```
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 ∈ ∅ is false, so we should be able to apply the disjunctive syllogism rule to h1 to infer the goal x ∈ B. To carry out this plan, we'll first have to prove x ∉ (∅ : Set U). 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	Tactic State in Infoview

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

    done
```

```
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	Tactic State in Infoview

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

```
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 ∈ ∅ is False. In other words, Lean knows that, by the definition of ∅, the statement x ∈ ∅ 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) :
    (∅ : Set U) ∪ B = B := by
  apply Set.ext
  fix x : U
  apply Iff.intro
  -- (→)
  assume h1 : x ∈ (∅ : Set U) ∪ B
  define at h1
  have h2 : x ∉ (∅ : Set U) := by
    by_contra h3
    define at h3  --Now h3 : False
    show False from h3
    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 ∈ ∅, 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) :
    (∅ : Set U) ∪ B = B := by
  apply Set.ext
  fix x : U
  apply Iff.intro
  -- (→)
  assume h1 : x ∈ (∅ : Set U) ∪ B
  define at h1
  have h2 : x ∉ (∅ : Set U) := by
    by_contra h3
    define at h3  --Now h3 : False
    show False from h3
    done
  disj_syll h1 h2  --Now h1 : x ∈ B
  show x ∈ B from h1
  -- (←)
```

Tactic State in Infoview

```
case h.mpr
U : Type
B : Set U
x : U
⊢ x ∈ B → x ∈ ∅ ∪ B
```

This direction of the biconditional proof is easier: once we introduce the assumption h1 : x ∈ B, our goal will be x ∈ ∅ ∪ B, which means x ∈ ∅ ∨ x ∈ B, and we can prove it with the proof Or.inr h1.

Lean File	Tactic State in Infoview

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

Goals accomplished 🎉

The second fact that was called "clear" in the proof from Example 3.6.2 was the equation C ∪ D = D ∪ C. This looks like an instance of the commutativity of the union operator. Let's prove that union is commutative.

Lean File	Tactic State in Infoview

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

```
U : Type
X Y : Set U
⊢ X ∪ Y = Y ∪ X
```

Once again, we begin with apply Set.ext, which converts the goal to ∀ (x : U), x ∈ X ∪ Y ↔ x ∈ Y ∪ X, and then fix x : U.

Lean File	Tactic State in Infoview

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

```
case h
U : Type
X Y : Set U
x : U
⊢ x ∈ X ∪ Y ↔ x ∈ Y ∪ X
```

By the definition of union, our current goal means x ∈ X ∨ x ∈ Y ↔ x ∈ Y ∨ x ∈ 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 ∨ Q ↔ Q ∨ 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. As soon as you type this, 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.

Lean File   Tactic State in Infoview

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

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. 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 Lean has applied a universal quantifier to the variables that were declared at the beginnings of the two theorems. When you declare variables in the statement of a theorem, it is understood that they can stand for anything of the appropriate type, so Lean treats them as implicitly universally quantified (see Section 3.1 of *HTPI*). Also, we used curly braces when we introduced the type `U`, so it is an implicit argument in both theorems and will not need to be specified when we apply the theorems.

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

Lean File   Tactic State in Infoview

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

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

Lean File

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

Tactic State in Infoview

```
case Existence
U : Type
⊢ ∃ A, ∀ (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 ∅ for `A`. It turns out that in this context, Lean is able to figure out which empty set we mean by ∅, so we don't need to specify the type.

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 ∅
```

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.

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
  -- Uniqueness
```

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 `∀ (B : Set U), C ∪ B = B` and `∀ (B : Set U), D ∪ B = B`, exactly as in the *HTPI* proof.

```
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
  -- 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
```

```
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.

```
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
  -- 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
```

```
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 ∪ C = C ∪ 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
-- 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
```

Since the statement ∃! (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:

To use a given of the form ∃! (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 ∀ (x_1 x_2 : U), P x_1 → P x_2 → x_1 = x2.

If you have a given h : ∃! (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 : ∀ (x_1 x_2 : U), P x_1 → P x_2 → 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
```

Tactic State in Infoview

```
U : Type
A B C : Set U
h1 : ∃ x, x ∈ A ∩ B
h2 : ∃ x, x ∈ A ∩ C
h3 : ∃! x, x ∈ A
⊢ ∃ x, 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.

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
```

Tactic State in Infoview

```
U : Type
A B C : Set U
h1 : ∃ x, x ∈ A ∩ B
h2 : ∃ x, x ∈ A ∩ C
h3 : ∃! x, 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, 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 ∈ A → c ∈ A → b = c, and therefore by two applications of modus ponens, h7 b c h4.left h5.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.left h5.left
```

Tactic State in Infoview

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

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-hand side of the equation that is replaced with the right-hand 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.symm` is a proof of `t = s`, you can use `rewrite [h.symm]`.) You can also apply the `rewrite` tactic to biconditional statements. If you have `h : P ↔ 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	Tactic State in Infoview

```
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
```

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

Now the right-hand 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),
    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)
```

You might want to compare the Lean proof above to the proof of this theorem as it appears in *HTPI*:

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. □

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 library_search. 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
```

Tactic State in Infoview

```
    case h
    U : Type
    X Y : Set U
    x : U
    ⊢ x ∈ X ∪ Y ↔ x ∈ Y ∪ X
```

To make the goal easier for the library_search tactic to recognize, we'll write out the definitions of the two sides of the biconditional. We use a variant 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. (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
```

Tactic State in Infoview

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

We have done these steps to give the `library_search` tactic the best chance of recognizing the goal as the commutative law for "or". Let's give it 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
  library_search
```

The blue squiggle indicates that the `library_search` tactic has produced an answer, which you will find in the Infoview pane: `exact or_comm' (x ∈ X) (x ∈ Y)`. 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 `library_search`'s answer as a shortened form of the tactic

```
show x ∈ X ∨ x ∈ Y ↔ x ∈ Y ∨ x ∈ X from or_comm' (x ∈ X) (x ∈ Y).
```

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 `library_search` 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 `library_search` in the proof with either `library_search`'s suggested `exact` tactic or the `show` tactic displayed above.

But what is `or_comm'`? The command `#check or_comm'` gives the result

```
or_comm' : ∀ (a b : Prop), a ∨ b ↔ b ∨ a.
```

So `or_comm'` is the same as `or_comm`, except that the arguments `a` and `b` are not implicit; that's why `library_search` had to specify values for those arguments in its answer. The `#check` command also produces a suggestion that we should use `or_comm` instead.

The `library_search` 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. Other tactics that may be helpful are `suggest` and `hint`.

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. To write formulas involving arithmetic operations, you should use the symbols `+` for addition, `-` for subtraction, `*` for multiplication, `/` for division, and `^` for exponentiation. 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 | y` to mean that `x` divides `y`, or `y` is divisible by `x`. The formal definition is that `x | y` means that there is an integer `k` such that `y = x * k`. For example, `3 | 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 ∧ B → C` as `A → B → C`.

Lean File   Tactic State in Infoview

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

```
⊢ ∀ (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 | b` and `b | c`. We also write out the definitions of our assumptions and the goal.

Lean File   Tactic State in Infoview

```
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
```

```
a b c : ℤ
h1 : ∃ c, b = a * c
h2 : ∃ c_1, c = b * c_1
⊢ ∃ c_1, 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   Tactic State in Infoview

```lean
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
```

```
a b c : ℤ
h1 : ∃ c, b = a * c
h2 : ∃ c_1, c = b * c_1
m : ℤ
h3 : b = a * m
n : ℤ
h4 : c = b * n
⊢ ∃ c_1, 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   Tactic State in Infoview

```lean
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
```

```
a b c : ℤ
h1 : ∃ c, b = a * c
h2 : ∃ c_1, c = b * c_1
m : ℤ
h3 : b = a * m
n : ℤ
h4 : c = a * m * n
⊢ ∃ c_1, 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   Tactic State in Infoview

```lean
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)
```

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

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 something like:

```
mul_assoc : ∀ (a b c : ?m.36649), a * b * c = a * (b * c).
```

The appearance of `?m...` here indicates that `mul_assoc` has implicit arguments. The implicit arguments in this case are a little complicated—of course, you can see them by typing `#check @mul_assoc`—but what they mean is that `mul_assoc` can be used to prove any statement of the form `∀ (a b c : U), a * b * c = a * (b * c)`, as long as `U` is a type that has an associative multiplication operation. In particular, `mul_assoc` can be used as a proof of `∀ (a b c : 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 : ∀ (a b c : ℕ), a * b * c = a * (b * c)`, `Int.mul_assoc : ∀ (a b c : 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 :
    ∀ (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)
  rewrite [mul_assoc a m n] at h4
```

Tactic State in Infoview

```
a b c : ℤ
h1 : ∃ c, b = a * c
h2 : ∃ c_1, c = b * c_1
m : ℤ
h3 : b = a * m
n : ℤ
h4 : c = a * (m * n)
⊢ 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 :
    ∀ (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)
rewrite [mul_assoc a m n] at h4
show c = a * (m * n) from h4
```

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*.

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 $6 \mid n$. $\qquad\square$

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
```

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 | n, and since the definition of this assumption is an existential statement, we immediately apply existential instantiation.

Lean File

Tactic State in Infoview

```
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
```

```
case mp
n : ℤ
h1 : ∃ c, 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

Tactic State in Infoview

```
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
  define
```

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

Tactic State in Infoview

```
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
  define
  apply Exists.intro (3 * k)
```

```
case mp.left
n : ℤ
h1 : ∃ c, 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
  define
  apply Exists.intro (3 * k)
  rewrite [←mul_assoc 2 3 k]
```

Tactic State in Infoview

```
case mp.left
n : ℤ
h1 : ∃ c, 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
  define
  apply Exists.intro (3 * k)
  rewrite [←mul_assoc 2 3 k]
  show n = 2 * 3 * k from h2
```

Tactic State in Infoview

```
case mp.right
n : ℤ
h1 : ∃ c, 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
  define
  apply Exists.intro (3 * k)
  rewrite [←mul_assoc 2 3 k]
  show n = 2 * 3 * k from h2
  define
  apply Exists.intro (2 * k)
  rewrite [←mul_assoc 3 2 k]
  show n = 3 * 2 * k from h2
```

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 | n ∧ 3 | n. We write out the definitions of 2 | n and 3 | 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
  have h2 := h1.left
  have h3 := 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
```

Tactic State in Infoview

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

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

Lean File	Tactic State in Infoview

```
theorem Theorem_3_4_7 :
    ∀ (n : Int), 6 | n ↔ 2 | n ∧ 3 | n := by
  fix n : Int
  apply Iff.intro
  -- (→)
  ...
  -- (←)
  assume h1
  have h2 := h1.left
  have h3 := 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
```

Goals accomplished 🎉

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 library_search`. Lean comes up with a justification: `exact Int.mul_sub 6 j k`, and `#check Int.mul_sub` produces the result

```
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	Tactic State in Infoview
| ```
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
``` | ```
n : ℤ
h1 : 2 | n ∧ 3 | n
h2 : ∃ c, n = 2 * c
h3 : ∃ c, 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	Tactic State in Infoview
| ```
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
``` | ```
n : ℤ
h1 : 2 | n ∧ 3 | n
h2 : ∃ c, n = 2 * c
h3 : ∃ c, 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 : ∀ (a : ?m.80342), a = a`. 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 ↔ a`, but also statements of the form `a ↔ 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 ↔ 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 : ℤ
h1 : 2 | n ∧ 3 | n
h2 : ∃ c, n = 2 * c
h3 : ∃ c, n = 3 * c
j : ℤ
h4 : n = 2 * j
k : ℤ
h5 : n = 3 * k
⊢ 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, n = 2 * c
h3 : ∃ c, 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 subtraction to the left of the multiplication rather than to the right. 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 library_search
```

Lean responds with `exact one_mul n`, and `#check one_mul` yields `one_mul : ∀ (a : ?m.73050),
1 * a = a`. So `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:

```
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 := 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
```

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 `A × B` is the type of ordered pairs `(a, b)`, where `a` has type `A` and `b` has type `B`. In other words, if you have `a : A` and `b : B`, then `(a, b)` is an object of type `A × B`. There is also notation for the first and second coordinates of an ordered pair. If `p` has type `A × B`, then `p.1` is the first coordinate of `p`, and `p.2` is the second coordinate. In other words, `p = (p.1, p.2)`. You can also use the notation `p.fst` for the first coordinate of `p` and `p.snd` for the second coordinate.

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 element 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` is a set whose elements are ordered pairs `(a, b)`, where `a` has type `A` and `b` has type `B`, then `R` has type `Set (A × B)`.

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

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 define as follows:

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

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:

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

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) :=
    {p : B × A | (p.2, p.1) ∈ R}
def comp {A B C : Type} (S : Set (B × C)) (R : Set (A × B)) :
    Set (A × C) := {p : A × C | ∃ (b : B), (p.1, b) ∈ R ∧ (b, p.2) ∈ S}
```

Definitions in Lean are introduced with the keyword `def`. In the definition of `Dom`, we have declared `A` and `B` to be implicit parameters. That means that 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. The definition declares that `Dom R` has type `Set A`, and then it defines it to be the set of all objects `a` of type `A` such that there is some `b` of type `B` with `(a, b) ∈ 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*:

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. $\mathrm{Dom}(R^{-1}) = \mathrm{Ran}(R)$.
3. $\mathrm{Ran}(R^{-1}) = \mathrm{Dom}(R)$.
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
```

```lean
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. Here is a natural way to start the proof:

```lean
theorem Theorem_4_2_5_4 {A B C : 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)
  define at h1
```

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

```lean
∀ (x : A × D), x ∈ comp T (comp S R) ↔ x ∈ comp (comp T S) R
```

The next step should be to introduce an arbitrary object of type `A × D`. We could just call this object `x`, but Lean let's us use a shortcut here. An object of type `A × D` must have the form of an ordered pair, where the first coordinate has type `A` and the second has type `D`. So Lean let's 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.)

After introducing the arbitrary ordered pair `(a, d)`, we must prove a biconditional statement, so we try to prove the two directions separately. For the left-to-right direction, we assume `h1 : (a, d) ∈ comp T (comp S R)`, and we must prove `(a, d) ∈ comp (comp T S) R`. When we ask Lean to write out the definition of the goal, the result is:

```lean
∃ b, ((a, d).fst, b) ∈ R ∧ (b, (a, d).snd) ∈ comp T S
```

Usually, the `define` tactic does a good job of writing out definitions, but in this case it has let us down a bit. The definition isn't wrong, but it's not written in the most convenient form. We can improve on the result by using the tactics `define : (a, d).fst` and `define : (a, d).snd`, which put the goal into this form:

```lean
∃ b, (a, b) ∈ R ∧ (b, d) ∈ comp T S
```

That solves the problem—this time. But we're going to have to write out the definition of composition of relations several more times in this proof; will we have to fix Lean's awkward definition every time?

A better idea is to teach Lean to do a better job of using the definition of composition. We can do that by proving a preliminary theorem before proving part 4 of Theorem 4.2.5:

```
theorem simp_comp
    {A B C : Type} (S : Set (B × C)) (R : Set (A × B)) (a : A) (c : C) :
    (a, c) ∈ comp S R ↔ ∃ (x : B), (a, x) ∈ R ∧ (x, c) ∈ S := by rfl
```

Now, any time we have relations R : Set (A × B) and S : (B × C), and objects a : A and c : C, the expression simp_comp S R a c will be a proof of the statement (a, c) ∈ comp S R ↔ ∃ (x : B), (a, x) ∈ R ∧ (x, c) ∈ S. And that means that the tactic rewrite [simp_comp S R a c] will change (a, c) ∈ comp S R to ∃ (x : B), (a, x) ∈ R ∧ (x, c) ∈ S, exactly as we wanted. In fact, as we've seen before, you can just write rewrite [simp_comp], and Lean will figure out how to apply the theorem simp_comp to rewrite some part of the goal.

Actually, it's more convenient to do the rewriting before the step apply Iff.intro—otherwise we'd have to do it twice, once for each direction of the biconditional proof. So now we start the proof like this:

```
theorem Theorem_4_2_5_4 {A B C : 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
  rewrite [simp_comp, simp_comp]
```

Why did we list simp_comp twice in the rewrite tactic? After the step fix (a, d) : A × D, the goal is

```
    (a, d) ∈ comp T (comp S R) ↔ (a, d) ∈ comp (comp T S) R
```

When we ask Lean to use the theorem simp_comp, it figures out that simp_comp T (comp S R) a d is a proof of the statement

```
    (a, d) ∈ comp T (comp S R) ↔ ∃ (x : C), (a, x) ∈ comp S R ∧ (x, d) ∈ T,
```

which can be used to rewrite the left-hand side of the goal. To rewrite the right-hand side, we need a different application of the simp_comp theorem, simp_comp (comp T S) R a d. So we have to ask Lean to apply the theorem a second time.

Here is the complete proof.

```
theorem Theorem_4_2_5_4 {A B C : 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
  rewrite [simp_comp, simp_comp]
    --Goal:  (∃ (x : C), (a, x) ∈ comp S R ∧ (x, d) ∈ T) ↔
    --            ∃ (x : B), (a, x) ∈ R ∧ (x, d) ∈ comp T S
  apply Iff.intro
  -- (→)
  assume 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
  rewrite [simp_comp] at h2
    --h2: (∃ (x : B), (a, x) ∈ R ∧ (x, c) ∈ S) ∧ (c, d) ∈ T
  obtain (b : B) (h3 : (a, b) ∈ R ∧ (b, c) ∈ S) from h2.left
  apply Exists.intro b --Goal:  (a, b) ∈ R ∧ (b, d) ∈ comp T S
  apply And.intro h3.left
  rewrite [simp_comp]  --Goal:  ∃ (x : C), (b, x) ∈ S ∧ (x, d) ∈ T
  show ∃ (x : C), (b, x) ∈ S ∧ (x, d) ∈ T from
    Exists.intro c (And.intro h3.right h2.right)
  -- (←)
  assume h1 : ∃ (x : B), (a, x) ∈ R ∧ (x, d) ∈ comp T S
  obtain (b : B) (h2 : (a, b) ∈ R ∧ (b, d) ∈ comp T S) from h1
  rewrite [simp_comp] at h2
  obtain (c : C) (h3 : (b, c) ∈ S ∧ (c, d) ∈ T) from h2.right
  apply Exists.intro c
  apply And.intro _ h3.right
  rewrite [simp_comp]
  show ∃ (x : B), (a, x) ∈ R ∧ (x, c) ∈ S from
    Exists.intro b (And.intro h2.left h3.left)
```

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