How To Prove It With Lean

Daniel J. Velleman

Table of contents

Preface 3
- About This Book 3
- About Lean 3
- Installing Lean 4

1 Sentential Logic 9

2 Quantificational Logic 10

Introduction to Lean 11
- A First Example 11
- Term Mode 12
- Tactic Mode 14
- Types 18

3 Proofs 21
- 3.1 & 3.2. Proofs Involving Negations and Conditionals 21
- 3.3. Proofs Involving Quantifiers 28

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.

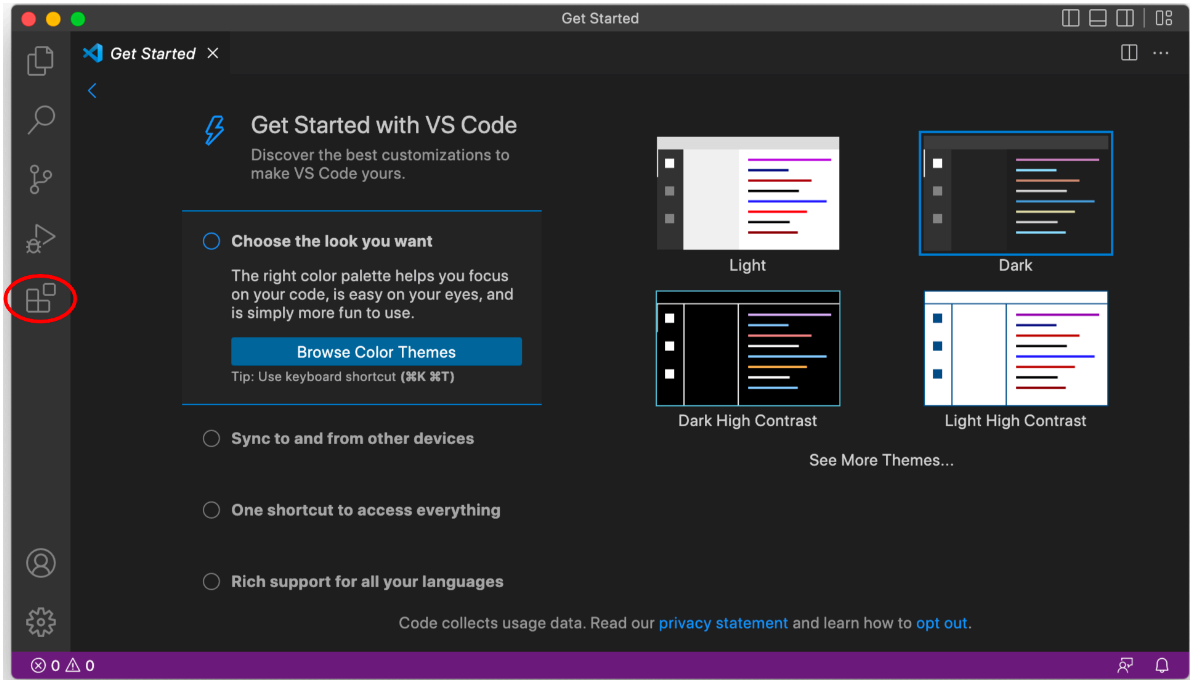
Of course, to make this possible you must type your proof in a format that Lean understands. Much of this book will be taken up with explaining how to write a proof so that Lean will understand it.

Installing Lean

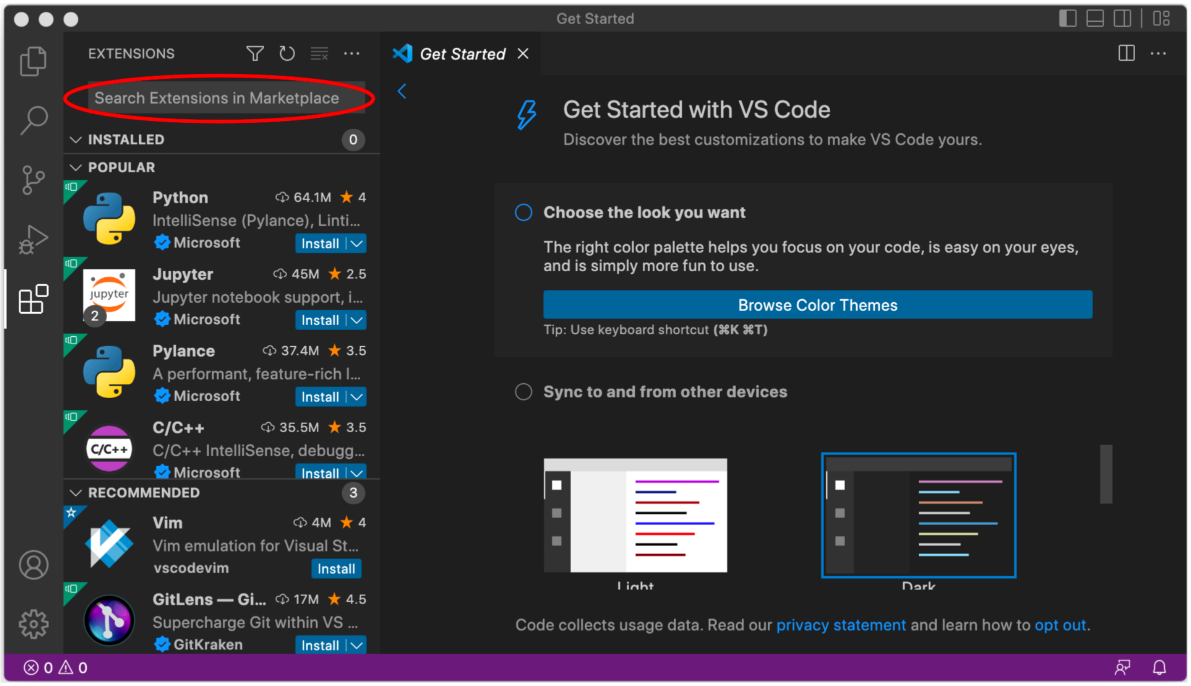
We will be using Visual Studio Code to run Lean, so you will need to install VS Code first. VS Code is free and can be downloaded here.

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.

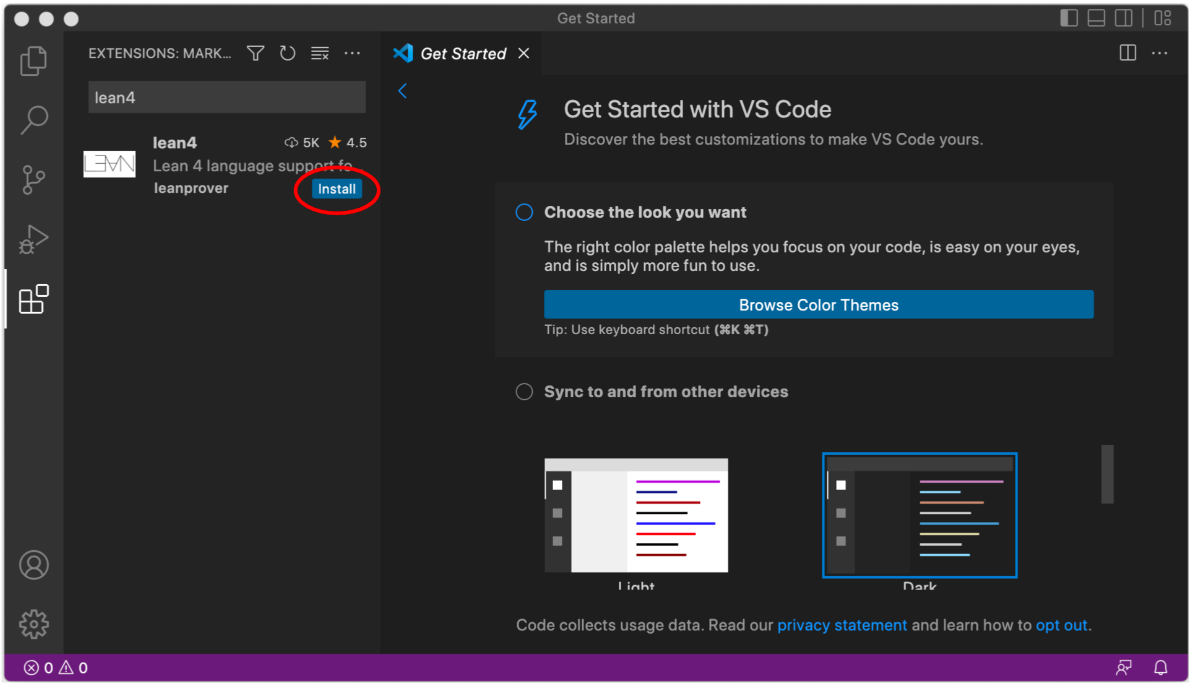
Now open VS Code. You should see a window that looks something like this:



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:

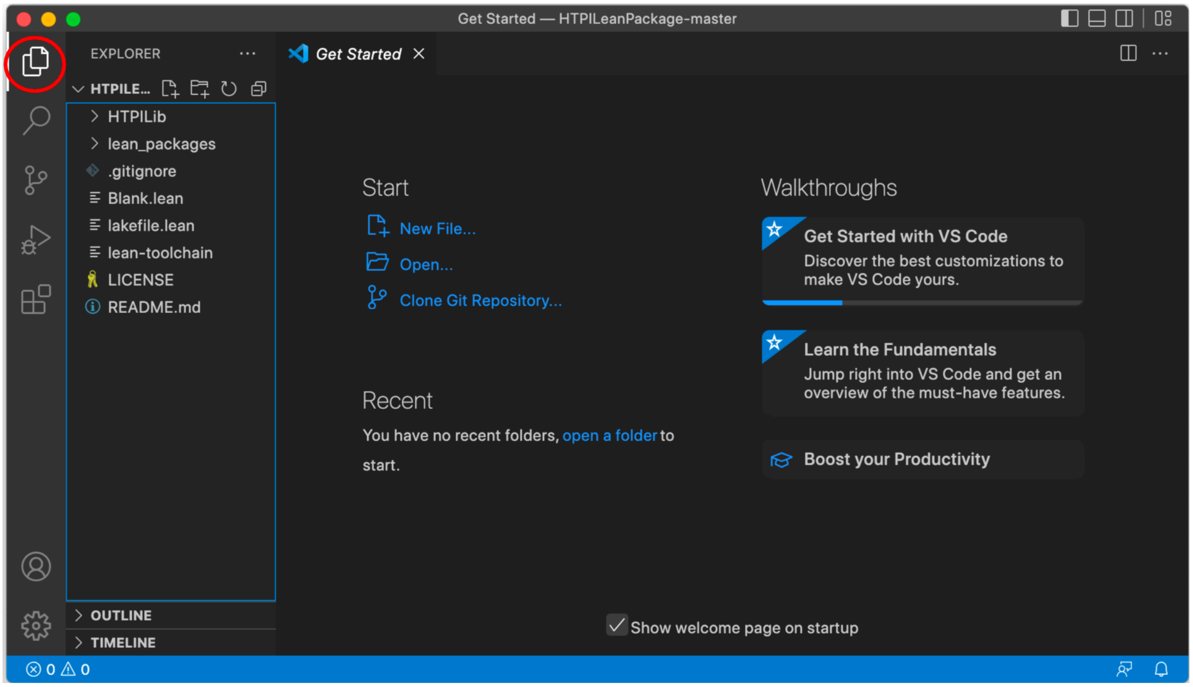


In the *Search Extensions in Marketplace* field, type “lean4”. VS Code should find the Lean 4 extension and display it:

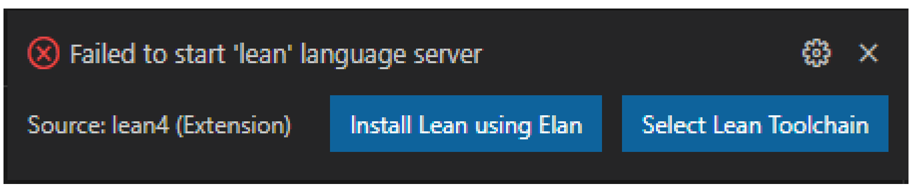


Click on "Install" to install the Lean 4 extension.

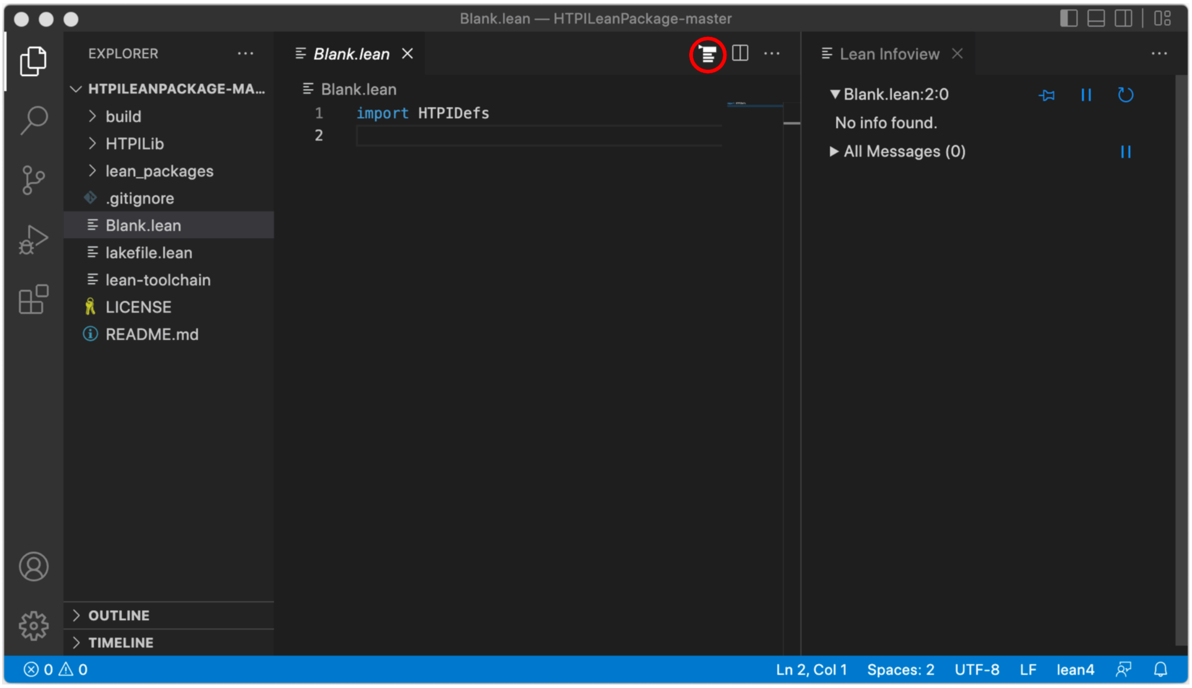
Next, in VS Code, select "Open Folder …" from the File menu and open the folder containing the HTPI Lean package that you downloaded earlier. Under the heading "Explorer" on the left side of the window, you should see a list of the files in the package. (If you don't see the list, try clicking on the *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
$\leftrightarrow$	if and only if

As we will see, Lean uses the same symbols, with the same meanings. 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:

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

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

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

$$A \mathbin{\triangle} B = (A \setminus B) \cup (B \setminus A) = \text{ the } \textit{symmetric difference} \text{ of } A \text{ and } B.$$

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:

Name		Equivalence	
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 $\mathcal{F}$ is a family of sets—that is, a set whose elements are sets—then

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

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

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

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 $\neg R \to (P \to \neg Q)$. □

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 for 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. 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
```

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, and also by marking where the proof ends. To do this, leave a blank line after the statement of the theorem and begin the next line with a tab; VS Code will indent two spaces. Then 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

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

  done
```

Tactic State in Infoview

```
P Q R : Prop
h1 : P → Q
h2 : Q → ¬R
⊢ R → ¬P
```

The red squiggle under `done` indicates that Lean knows that the proof isn't done. The tactic state in the Infoview pane is very similar to the lists of givens and goals that are used in *HTPI*. The tactic state above says that `P`, `Q`, and `R` stand for propositions, and we have two givens, `h1 : P → Q` and `h2 : Q → ¬R`. The symbol `⊢` in the last line labels the goal, `R → ¬P`.

From the hypotheses `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 tab to indent two spaces and then `contrapos` to tell Lean that you want to replace the goal with its contrapositive. (You won't have to type tab to indent later lines; VS Code maintains the same indenting until you delete the tab at the beginning of a line to return to unindented text.) As soon as you type `contrapos`, Lean will update the tactic state to reflect the change in the goal. You should now see this:

Lean File

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

Tactic State in Infoview

```
P Q R : Prop
h1 : P → Q
h2 : Q → ¬R
⊢ P → ¬R
```

If you want to make your proof a little more readable, you could add a comment saying that the goal has been changed to `P → ¬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 the new given `h3 : P`, and it knows, without having to be told, that the goal should now be `¬R`:

Lean File

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

Tactic State in Infoview

```
P Q R : Prop
h1 : P → Q
h2 : Q → ¬R
h3 : P
⊢ ¬R
```

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

Lean File

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

Tactic State in Infoview

```
P Q R : Prop
h1 : P → Q
h2 : Q → ¬R
h3 : P
h4 : Q
⊢ ¬R
```

Finally, to complete the proof, we can infer the goal `¬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":

Lean File

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

Tactic State in Infoview

```
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.

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`. 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:

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`

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

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

  done
```

Tactic State in Infoview

```
B C : Set ℕ
a : ℕ
h1 : a ∈ B
h2 : ¬a ∈ B \ C
⊢ a ∈ C
```

The second line of this theorem statement declares that the variables `B` and `C` stand for sets of natural numbers, and `a` stands for a natural number. The third line states the two hypotheses of the theorem, `a ∈ 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

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

Tactic State in Infoview

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

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

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

Tactic State in Infoview

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

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

Lean File

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

Tactic State in Infoview

```
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

```
  ⋮
⊢ P → Q
```

Tactic State After Using Strategy

```
  ⋮
h : P
⊢ Q
```

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

Tactic State Before Using Strategy

```
  ⋮
⊢ P → Q
```

Tactic State After Using Strategy

```
  ⋮
h : ¬Q
⊢ ¬P
```

Section 3.2 gives two strategies for using givens of the form `P → 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`, and `double_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`, and `double_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`. 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 `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

```
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 _
```

Tactic State in Infoview

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

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

```
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
```

Tactic State in Infoview

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

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

Lean File

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

Tactic State in Infoview

```
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 a 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, $\forall x\, P(x) \wedge Q(x)$ is interpreted as $(\forall x\, P(x)) \wedge Q(x)$; if you want the quantifier $\forall x$ to apply to the entire statement $P(x) \wedge Q(x)$ you must use parentheses and write $\forall x(P(x) \wedge Q(x))$. The convention in Lean is exactly the opposite: a quantifier applies to as much as possible. Thus, Lean will interpret `∀ (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

```
  ⋮
⊢ ∀ (x : U), P x
```

Tactic State After Using Strategy

```
  ⋮
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

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

Tactic State in Infoview

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

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

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

`quant_neg` Tactic		
`¬∀ (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`
𝒫	`\powerset`

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

Lean File

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

Tactic State in Infoview

```
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

```
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
```

Tactic State in Infoview

```
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

```
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
```

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
⊢ 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 blank can be filled in with anything of type `U`. 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 blank must be filled in with `y`.

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
  have h5 : y ∈ B ∪ C := h1 h3
```

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
h5 : y ∈ B ∪ C
⊢ y ∈ C
```

Now we're almost done, because we can combine `h4` and `h5` to reach our goal. To see how, we first write out the definition of `h5`.

Lean File

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

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	Tactic State in Infoview

<pre>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</pre>
	<pre>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</pre>

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 proof, 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	Tactic State in Infoview

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

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

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

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

Lean File

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

Tactic State in Infoview

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

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

Lean File

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

Tactic State in Infoview

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

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

Lean File

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

Tactic State in Infoview

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

Now `h5 h7` is a proof of `¬Q b` and `h6 h7` is a proof of `Q b`, so `(h5 h7) (h6 h7)` is a proof of `False`.

Lean File

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

Tactic State in Infoview

```
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 `⋃₀`, which is entered by typing `\U0` (that is, backslash-capital U-zero). For an intersection of a family of sets, Lean uses `⋂₀`, typed as `\I0`.

Lean File

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

Tactic State in Infoview

```
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

```
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
```

Tactic State in Infoview

```
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)) : ⋃₀F ⊆ B → F ⊆ 𝒫 B := by
  assume h1 : ⋃₀F ⊆ 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 : ⋃₀F ⊆ 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)) : ⋃₀F ⊆ B → F ⊆ 𝒫 B := by
  assume h1 : ⋃₀F ⊆ 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 : ⋃₀F ⊆ 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)) : ⋃₀F ⊆ B → F ⊆ 𝒫 B := by
  assume h1 : ⋃₀F ⊆ 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 ∈ ⋃₀F → 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 ∈ ⋃₀F`. So we use the `apply h1 _` tactic.

Lean File

```
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
  fix x : Set U
  assume h2 : x ∈ F
  define
  fix y : U
  assume h3 : y ∈ x
  define at h1
  apply h1 _
```

Tactic State in Infoview

```
U : Type
B : Set U
F : Set (Set U)
h1 : ∀ {a : U},
  a ∈ ⋃₀F → a ∈ B
x : Set U
h2 : x ∈ F
y : U
h3 : y ∈ x
⊢ y ∈ ⋃₀F
```

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

Lean File

```
theorem Example_3_3_5 (U : Type) (B : Set U)
(F : Set (Set U)) : ⋃₀F ⊆ B → F ⊆ 𝒫 B := by
  assume h1 : ⋃₀F ⊆ B
  define
  fix x : Set U
  assume h2 : x ∈ F
  define
  fix y : U
  assume h3 : y ∈ x
  define at h1
  apply h1 _
  define
```

Tactic State in Infoview

```
U : Type
B : Set U
F : Set (Set U)
h1 : ∀ {a : U},
  a ∈ ⋃₀F → a ∈ B
x : Set U
h2 : x ∈ F
y : U
h3 : y ∈ x
⊢ ∃ (a : Set U),
  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 _`.

Lean File

```
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
  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 _
```

Tactic State in Infoview

```
U : Type
B : Set U
F : Set (Set U)
h1 : ∀ {a : U},
  a ∈ ⋃₀F → 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.

Lean File

```
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
  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
```

Tactic State in Infoview

```
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. □