How To Prove It With Lean

Daniel J. Velleman

Table of contents

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.

On every page of this book, 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

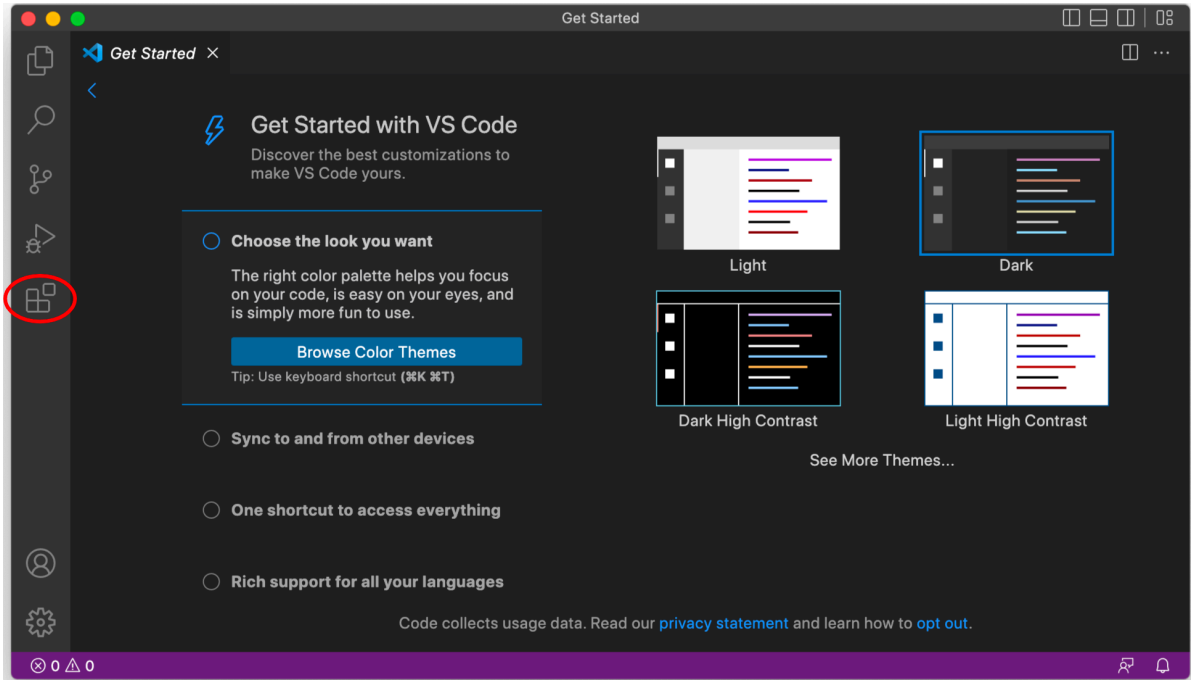
finish the proof, and it will display that information for you. That can keep you moving in the right direction as you are figuring out a proof. And sometimes Lean can fill in small details of the proof for you.

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.

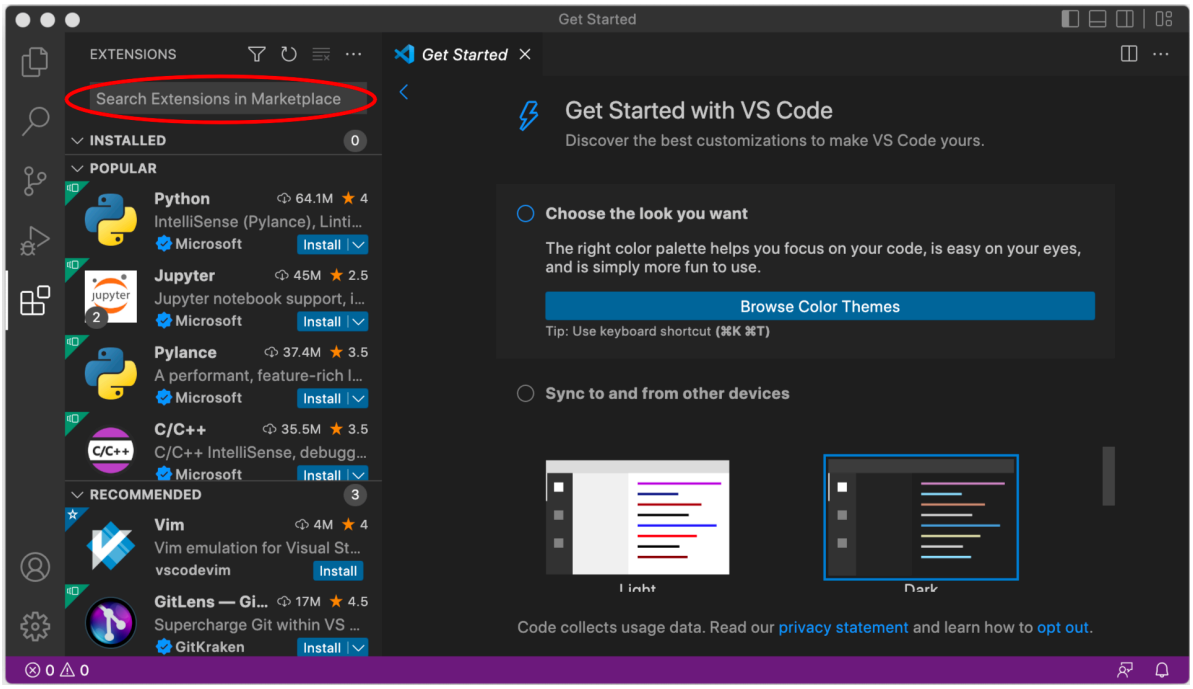
Installing Lean

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 here (***add link***).

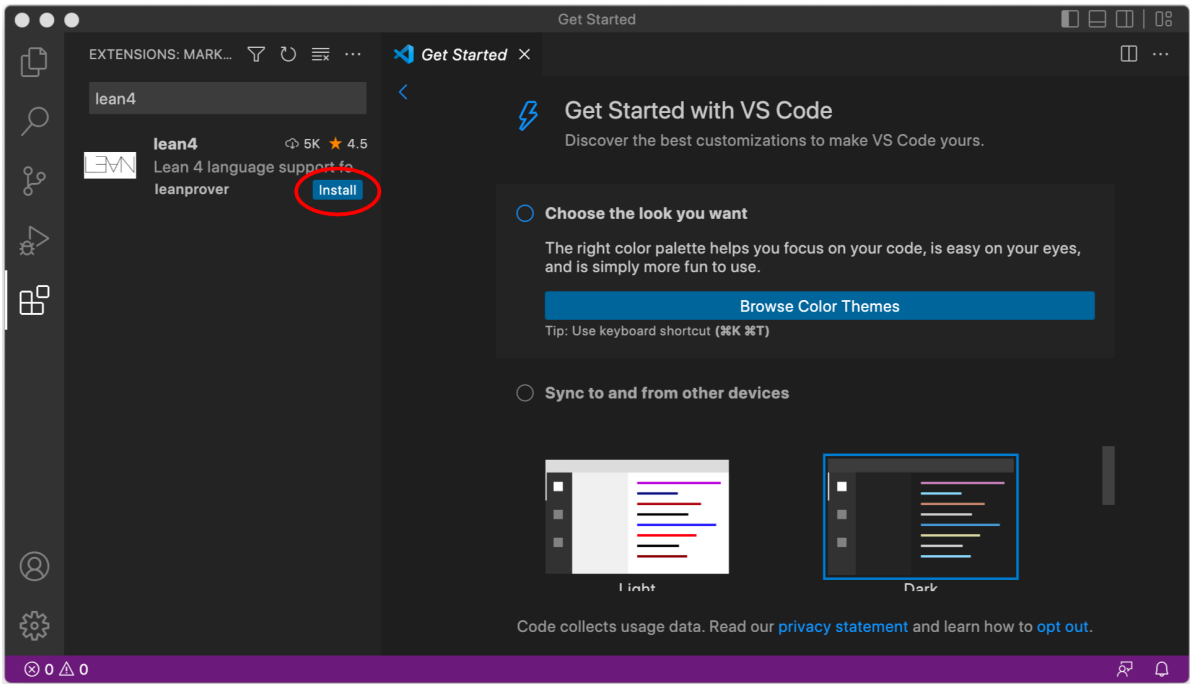
When you first start VS Code, you should see a window that looks something like this:



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.

Further steps: Open my Lean package in VS Code. Should get a message saying "Failed to start 'lean' language server." Click on the "Install Lean using Elan" button, and Lean should be installed. Will Infoview automatically open at that point?

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			
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*.

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,

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

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

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

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

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

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. [*** further instructions for opening an empty file with import HTPIdefs***]

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 `:=`. [Maybe: Tell user to add "done"?] 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 type `by`, 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!!
```

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

The red squiggle under `by` indicates that Lean knows that the proof is incomplete. 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`.

Although it is not necessary, we find it helpful to set off a tactic proof from the surrounding text by indenting it. To indent the proof, begin the next line with a tab; VS Code will indent two spaces. 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.

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 `contrapos` 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
```

```
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 the next line. 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

Tactic State in Infoview

```
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
```

```
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

Tactic State in Infoview

```
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
```

```
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 `by`, and the tactic state will say "Goals accomplished":

Lean File

Tactic State in Infoview

```
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
```

```
-Goals accomplished
```

Congratulations! You've written your first proof in tactic mode. 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

Tactic State in Infoview

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

```
A B : Set
a :
h1 : a ∈ A
h2 : ¬a ∈ A \ B
⊢ a ∈ B
```

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

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
(A B : Set Nat) (a : Nat)
(h1 : a ∈ A) (h2 : a ∈ A \ B) : a ∈ B := !!by!!
  define at h2       --Now h2 : ¬(a ∈ A ∧ ¬a ∈ B)
```

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

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

Lean File

Tactic State in Infoview

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

```
A B : Set
a :
h1 : a ∈ A
h2 : a ∈ A → a ∈ B
 a ∈ B
```

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
(A B : Set Nat) (a : Nat)
(h1 : a ∈ A) (h2 : a ∈ A \ B) : a ∈ B := by
  define at h2       --Now h2 : ¬(a ∈ A ∧ ¬a ∈ B)
  demorgan at h2     --Now h2 : ¬a ∈ A ∨ a ∈ B
  conditional at h2  --Now h2 : a ∈ A → a ∈ B
  show a ∈ B from h2 h1
```

```
-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) (A B : Set U) (a : U)
(h1 : a ∈ A) (h2 : a ∈ A \ B) : a ∈ B := !!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 `A` and `B` 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

The first proof strategies in *How To Prove It* are presented in Section 3.2, so that is where we begin.

3.2. Proofs Involving Negations and Conditionals

When stating proof strategies, can give before and after versions of tactic state:

Tactic state before using strategy

Tactic state after using strategy

```
-
  P → Q
```

```
-
h : P
  Q
```