How To Prove It With Lean

Daniel J. Velleman

Table of contents

Table of contents

Preface

About This Book

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

HTPI explains a systematic approach to constructing mathematical proofs. The purpose of this book is to show you how to use a computer software package called *Lean* to help you master the techniques presented in *HTPI*. Lean is free software that is available for Windows, MacOS, and Unix computers. To get the most out of this book, you will need to download and install Lean on your computer. We will explain how to do that below.

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

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

About Lean

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

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

Installing Lean

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

You will also need the Lean package that accompanies this book, which can be downloaded from https://github.com/djvelleman/HTPILeanPackage. After following the link, click on the green "Code" button and, in the pop-up menu, select "Download ZIP". Open the downloaded zip file to create a folder containing the HTPI Lean package. You can put this folder wherever you want on your computer.

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

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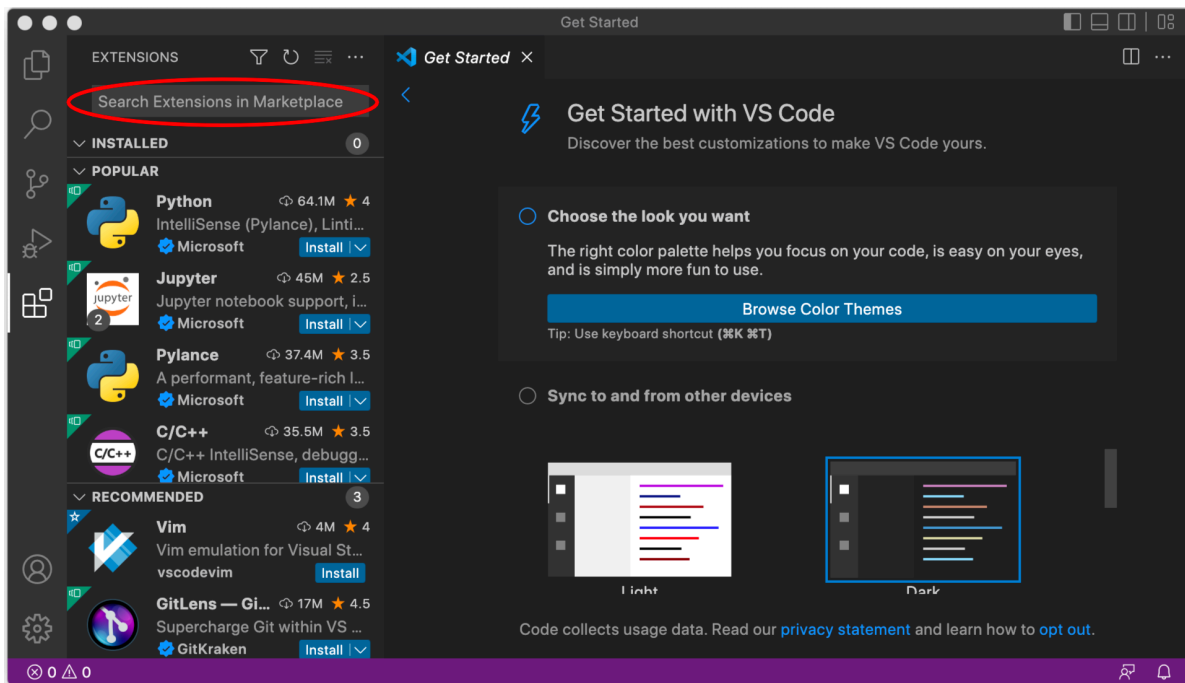
In the *Search Extensions in Marketplace* field, type "lean4". VS Code should find the Lean 4

extension and display it:

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

Next, in VS Code, select "Open Folder ..." from the File menu and open the folder containing the HTPI Lean package that you downloaded earlier. Under the heading "Explorer" on the left side of the window, you should see a list of the files in the package. (If you don't see the list, try clicking on the *Explorer* icon, circled in red below.)
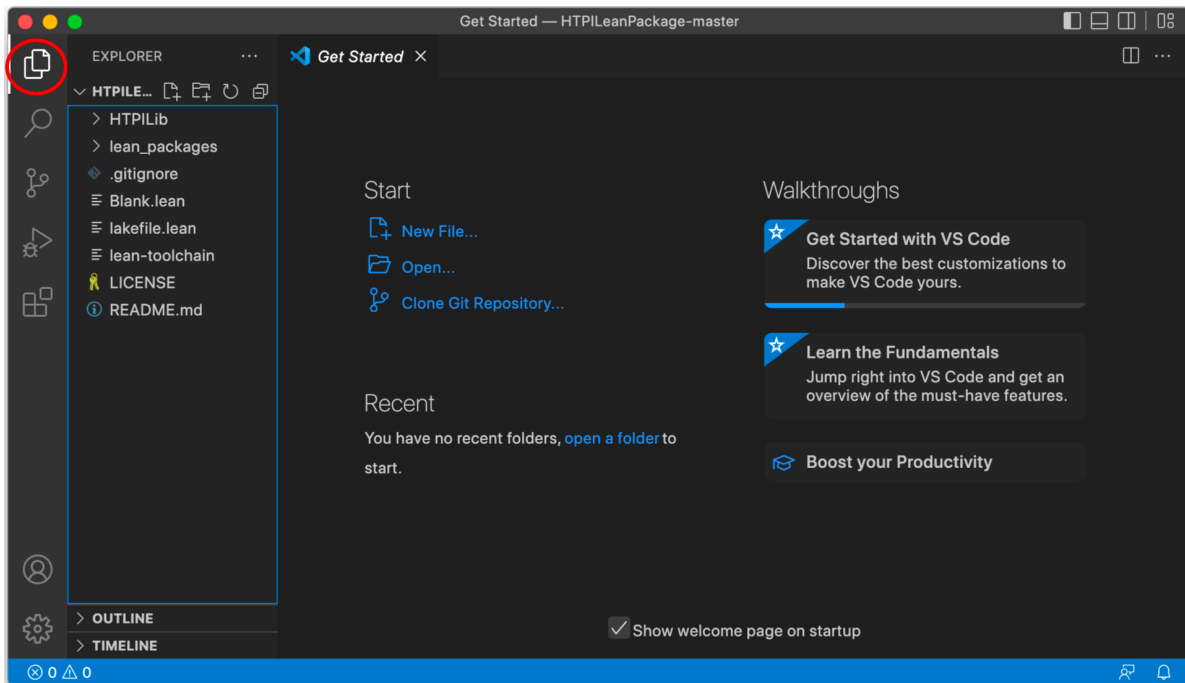
Click on the file "Blank.lean" in the file list. You should see a warning that VS Code failed to start the 'lean' language server:
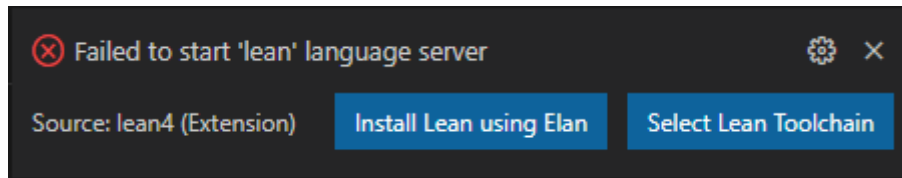
Click on the "Install Lean using Elan" button, and the Lean server should be installed. This may take a while, and there may be messages asking you to do things. If anything goes wrong, try quiting VS Code and restarting. Eventually your window should look like this:
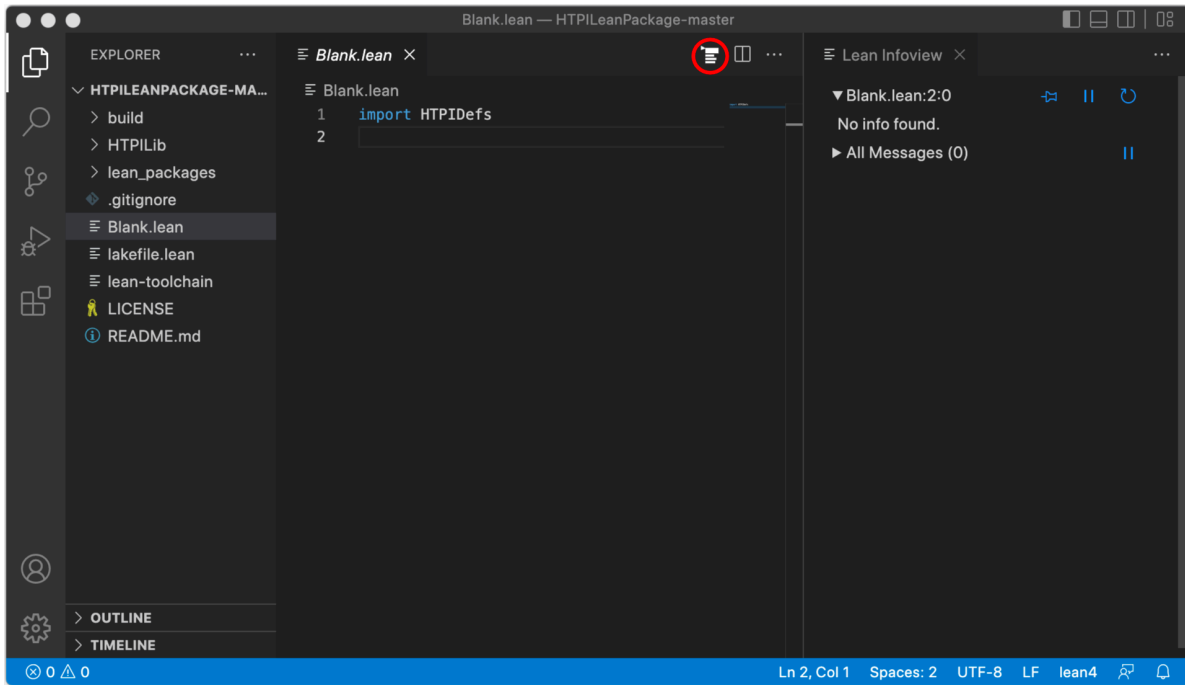
If you don't see the Infoview pane on the right side of the window, click on the icon circled in red in the image above, and the Infoview pane should appear.

Your installation is now complete.

1 Sentential Logic

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

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

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

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

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

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

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

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

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

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

2 Quantificational Logic

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

$\forall x\, P(x)$ means "for all x, $P(x)$,"

and

$\exists x\, P(x)$ means "there exists some x such that $P(x)$."

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

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

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

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

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

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

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

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

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

Introduction to Lean

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

A First Example

We'll start with Example 3.2.4 in *How To Prove It*. Here is how the theorem and proof in that example appear in *HTPI* (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. (If you are reading this book online, then Lean examples like the one below will appear in gray boxes. You can copy the example to your clipboard by clicking in the upper-right corner of the box, and then you can paste it into a file in VS Code to try it out.)

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

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 step of the proof, we combine the new h4 with h2 to infer ¬Q. There is no need to give this statement an identifier, because it completes the proof. In the proof in *HTPI*, there are a couple of final sentences explaining *why* this completes the proof, but Lean doesn't require this explanation.

Term Mode

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

If you have not yet installed Lean on your computer, go back and follow the instructions for installing it now. Then in VS Code, open the folder containing the HTPI Lean Package that you downloaded, and click on the file Blank.lean. The file starts with the line import HTPIDefs. Click on the blank line at the end of the file; this is where you will be typing your first proofs.

Now type in the following theorem and proof:

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

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

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

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

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

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

This indicates that Lean has detected an error in the proof. Lean always indicates errors by putting a red squiggle under the offending text. Lean also puts a message in the Lean Infoview pane explaining what the error is. (If you don't see the Infoview pane, choose "Command Palette …" in the "View" menu, and then type "Lean" in the text box that appears. You will see a list of commands that start with "Lean". Click on "Lean 4: Infoview: Toggle" to make the Infoview pane appear.) In this case, the message is `unknown identifier 'f'`. The message is introduced by a heading, in red, that identifies the file, the line number, and the character position on that line where the error appears. If you change `f` back to `h`, the red squiggle and error message go away.

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

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

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

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

We'll try one more proof in term mode:

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

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

Tactic Mode

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

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

Lean is now waiting for you to type a proof in term mode. To switch to tactic mode, type `by` after `:=`. We find it helpful to set off a tactic proof from the surrounding text by indenting it

with one tab, and also by marking where the proof ends. To do this, leave a blank line after the statement of the theorem, adjust the indenting to one tab, and type `done`. You will type your proof between the statement of the theorem and the line containing `done`, so click on the blank line between them to position the cursor there. Lean can be fussy about indenting; it will be important to indent all steps of the proof by the same amount.

One of the advantages of tactic mode is that Lean displays, in the Lean Infoview pane, information about the status of the proof as your write it. As soon as you position your cursor on the blank line, Lean displays what it calls the "tactic state" in the Infoview pane. Your screen should look like this:

Lean File   Tactic State in Infoview

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

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

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

Lean File   Tactic State in Infoview

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

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

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 "No goals" to indicate that there is nothing left to prove:

Lean File

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

Tactic State in Infoview

```
No goals
```

Congratulations! You've written your first proof in tactic mode. If you move your cursor around in the proof, you will see that Lean always displays in the Infoview the tactic state at the point in the proof where the cursor is located. Try clicking on different lines of the proof to see how the tactic state changes over the course of the proof. If you want to try another

example, you could try typing in the first example in this chapter. You will learn the most from this book if you continue to type the examples into Lean and see for yourself how the tactic state gets updated as the proof is written.

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

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

Symbol	How To Type It	Plain Text Alternative
¬	`\not` or `\n`	`Not`
∧	`\and`	`/\`
∨	`\or` or `\v`	`\/`
→	`\to` or `\r` or `\imp`	`->`
↔	`\iff` or `\lr`	`<->`

Lean has conventions that it follows to interpret a logical statement when there are not enough parentheses to indicate how terms are grouped in the statement. For our purposes, the most important of these conventions is that `P → Q → R` is interpreted as `P → (Q → R)`, not `(P → Q) → R`. The reason for this is simply that statements of the form `P → (Q → R)` come up much more often in proofs than statements of the form `(P → Q) → R`. (Lean also follows this "grouping-to-the-right" convention for ∧ and ∨, although this makes less of a difference, since these connectives are associative.) Of course, when in doubt about how to type a statement, you can always put in extra parentheses to avoid confusion.

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

Logical Law	Tactic		Transformation	
Contrapositive Law	`contrapos`	`P → Q`	is changed to	`¬Q → ¬P`
De Morgan's Laws	`demorgan`	`¬(P ∧ Q)`	is changed to	`¬P ∨ ¬Q`
		`¬(P ∨ Q)`	is changed to	`¬P ∧ ¬Q`
		`P ∧ Q`	is changed to	`¬(¬P ∨ ¬Q)`
		`P ∨ Q`	is changed to	`¬(¬P ∧ ¬Q)`
Conditional Laws	`conditional`	`P → Q`	is changed to	`¬P ∨ Q`
		`¬(P → Q)`	is changed to	`P ∧ ¬Q`
		`P ∨ Q`	is changed to	`¬P → Q`
		`P ∧ Q`	is changed to	`¬(P → ¬Q)`
Double Negation Law	`double_neg`	`¬¬P`	is changed to	`P`
Biconditional Negation Law	`bicond_neg`	`¬(P ↔ Q)`	is changed to	`¬P ↔ Q`
		`P ↔ Q`	is changed to	`¬(¬P ↔ Q)`

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

Types

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

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

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

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

Lean File

```
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. (Note that Lean occasionally writes things slightly differently in the tactic state. In this case, Lean has written ℕ instead of Nat, and ¬a ∈ B \ C instead of a ∉ B \ 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
```

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

Lean File

```
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   Tactic State in Infoview

```
theorem Example_3_2_5_simple                        No goals
    (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
```

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

$$\vdots$$
`⊢ P → Q`

Tactic State After Using Strategy

$$\vdots$$
`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

$$\vdots$$
`⊢ P → Q`

Tactic State After Using Strategy

$$\vdots$$
`h : ¬Q`
`⊢ ¬P`

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

To use a given of the form `P → Q`:

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

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

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

Section 3.2 of *HTPI* offers two strategies for proving negative goals:

To prove a goal of the form `¬P`:

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

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

Tactic State Before Using Strategy   Tactic State After Using Strategy

```
     ⋮                                           ⋮
 ⊢ ¬P                                        h : P
                                             ⊢ False
```

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

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

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

Q	¬Q	(Q	→	False)
F	T	F	T	F
T	F	T	F	F

Thinking of `h2 : ¬Q` as meaning `h2 : Q → False`, we can combine it with `h1 : Q` using modus ponens to deduce `False`. In other words, `h2 h1` is a proof of `False`.

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

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

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

Lean File

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

Tactic State in Infoview

```
P Q R : Prop
h : P → 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
  done
```

Tactic State in Infoview

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

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

Lean File

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

Tactic State in Infoview

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

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

Lean File

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

Tactic State in Infoview

```
No goals
```

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

To use a given of the form ¬P:

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

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

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

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

Lean File

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

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
  done
```

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
  done
```

Tactic State in Infoview

```
No goals
```

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

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

Lean File

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

  done
```

Tactic State in Infoview

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

Tactic State in Infoview

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

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
  done
```

Tactic State in Infoview

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

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

Lean File   Tactic State in Infoview

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

```
No goals
```

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.

Exercises

Fill in proofs of the following theorems. All exercises are taken from *HTPI*.

1.
```
theorem Exercise_3_2_1a (P Q R : Prop)
    (h1 : P → Q) (h2 : Q → R) : P → R := by

  done
```

2.
```
theorem Exercise_3_2_1b (P Q R : Prop)
    (h1 : ¬R → (P → ¬Q)) : P → (Q → R) := by

  done
```

3.
```
theorem Exercise_3_2_2a (P Q R : Prop)
    (h1 : P → Q) (h2 : R → ¬Q) : P → ¬R := by

  done
```

4.
```
theorem Exercise_3_2_2b (P Q : Prop)
    (h1 : P) : Q → ¬(Q → ¬P) := by

  done
```

3.3. Proofs Involving Quantifiers

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

To use a predicate in Lean, you must tell Lean the type of objects to which it applies. If `U` is a type, then `Pred U` is the type of predicates that apply to objects of type `U`. If `P` has type `Pred U` (that is, `P` is a predicate applying to objects of type `U`) and `x` has type `U`, then to apply `P` to `x` we just write `P x` (with a space but no parentheses). Thus, if we have `P : Pred U` and `x : U`, then `P x` is an expression of type `Prop`. That is, `P x` is a proposition, and its meaning is that `x` has the property represented by the predicate `P`.

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

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

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

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

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

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

Tactic State Before Using Strategy

```
    ⋮
⊢ ∀ (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

  done
```

Tactic State in Infoview

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

To use the givens h1 and h2, we will probably want to use universal instantiation. But to do that we would need an object of type U to plug in for x in h1 and h2, and there is no object of

type `U` in the tactic state. So at this point, we can't apply universal instantiation to `h1` and `h2`. We should watch for an object of type `U` to come up in the course of the proof, and consider applying universal instantiation if one does. Until then, we turn our attention to the goal.

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

quant_neg Tactic		
`¬∀ (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
  done
```

Tactic State in Infoview

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

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

Lean File

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

Tactic State in Infoview

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

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

Lean File

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

Tactic State in Infoview

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

We're almost done, because the goal now follows easily from `h3` and `h4`. If we use the contrapositive law to rewrite `h3` as `Q y → ¬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
  done
```

Tactic State in Infoview

```
No goals
```

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

Symbol	How To Type It
∈	\in
∉	\notin or \inn
⊆	\sub
⊄	\subn
=	=

Symbol	How To Type It
≠	\ne
∪	\union or \cup
∩	\inter or \cap
\	\\
△	\bigtriangleup
∅	\emptyset
𝒫	\powerset

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

Lean File

Tactic State in Infoview

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

  done
```

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

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

Lean File

Tactic State in Infoview

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

```
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 those funny double braces rather than parentheses? We'll return to that question shortly. The difference doesn't affect our next steps, which are to introduce an arbitrary object `y` of type `U` and assume `y ∈ A`.

Lean File

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

Tactic State in Infoview

```lean
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

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

Tactic State in Infoview

```lean
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

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

Tactic State in Infoview

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

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

Lean File

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

Tactic State in Infoview

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

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

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

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

Lean File

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

Tactic State in Infoview

```
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

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

Tactic State in Infoview

```
No goals
```

Next we turn to strategies for working with existential quantifiers.

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   Tactic State After Using Strategy

```
    ⋮                                           ⋮
  a : U                                       a : U
  ⊢ ∃ (x : U), P x                            ⊢ P a
```

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

To use a given of the form ∃ (x : U), P x:

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

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

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

Lean File

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

  done
```

Tactic State in Infoview

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

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

Lean File

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

Tactic State in Infoview

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

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

Lean File

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

Tactic State in Infoview

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

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

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

Lean File

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

Tactic State in Infoview

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

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

Lean File

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

Tactic State in Infoview

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

Now look at `h5` and `h6`. They show that `P a` leads to contradictory conclusions, `¬Q b` and `Q`

b. This means that `P a` must be false. We finally know what value of `x` to use to prove the goal.

Lean File

Tactic State in Infoview

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

```
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

Tactic State in Infoview

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

```
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

Tactic State in Infoview

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

```
No goals
```

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

Lean File

Tactic State in Infoview

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

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

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

Lean File

Tactic State in Infoview

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

```
U : Type
B : Set U
F : Set (Set U)
h1 : ⋃₀F ⊆ B
⊢ ∀ ⦃a : Set U⦄,
    a ∈ F → a ∈ 𝒫 B
```

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

Lean File

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

Tactic State in Infoview

```
U : Type
B : Set U
F : Set (Set U)
h1 : U₀F ⊆ B
x : Set U
h2 : x ∈ F
⊢ ∀ ⦃a : U⦄,
    a ∈ x → a ∈ B
```

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

Lean File

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

Tactic State in Infoview

```
U : Type
B : Set U
F : Set (Set U)
h1 : U₀F ⊆ B
x : Set U
h2 : x ∈ F
y : U
h3 : y ∈ x
⊢ y ∈ B
```

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

Lean File

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

Tactic State in Infoview

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

Now we see that we can try to use `h1` to reach our goal. Indeed, `h1 _` would be a proof of the goal if we could fill in the blank with a proof of $y \in \cup_0 F$. So we use the `apply h1 _` tactic.

Lean File

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

Tactic State in Infoview

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

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

Tactic State in Infoview

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

Our goal is now an existential statement, so we look for a value of `t` that will make the statement `t ∈ F ∧ y ∈ t` true. The givens `h2` and `h3` tell us that `x` is such a value, so as described earlier our next tactic should be `apply Exists.intro x _`.

Lean File

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

Tactic State in Infoview

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

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

Lean File

```
theorem Example_3_3_5 (U : Type) (B : Set U)
    (F : Set (Set U)) : U₀F ⊆ B → F ⊆ 𝒫 B := by
  assume h1 : U₀F ⊆ B
  define
  fix x : Set U
  assume h2 : x ∈ F
  define
  fix y : U
  assume h3 : y ∈ x
  define at h1
  apply h1 _
  define
  apply Exists.intro x _
  show x ∈ F ∧ y ∈ x from And.intro h2 h3
  done
```

Tactic State in Infoview

```
No goals
```

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

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

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

Exercises

1.
```
theorem Exercise_3_3_1
    (U : Type) (P Q : Pred U) (h1 : ∃ (x : U), P x → Q x) :
    (∀ (x : U), P x) → ∃ (x : U), Q x := by

  done
```

2.
```
theorem Exercise_3_3_8 (U : Type) (F : Set (Set U)) (A : Set U)
    (h1 : A ∈ F) : A ⊆ ⋃₀F := by

  done
```

3.
```
theorem Exercise_3_3_9 (U : Type) (F : Set (Set U)) (A : Set U)
    (h1 : A ∈ F) : ⋂₀F ⊆ A := by

  done
```

4.
```
theorem Exercise_3_3_10 (U : Type) (B : Set U) (F : Set (Set U))
    (h1 : ∀ (A : Set U), A ∈ F → B ⊆ A) : B ⊆ ⋂₀F := by

  done
```

5.
```
theorem Exercise_3_3_13 (U : Type)
    (F G : Set (Set U)) : F ⊆ G → ⋂₀G ⊆ ⋂₀F := by

  done
```

3.4. Proofs Involving Conjunctions and Biconditionals

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

To prove a goal of the form `P ∧ Q`:

  Prove `P` and `Q` separately.

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

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

The strategy for a given that is a conjunction is similar.

To use a given of the form `P ∧ Q`:

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

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

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

Lean File

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

  done
```

Tactic State in Infoview

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

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

Lean File

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

Tactic State in Infoview

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

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

Lean File

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

Tactic State in Infoview

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

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

Lean File

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

Tactic State in Infoview

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

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

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

Lean File

Tactic State in Infoview

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

    done
  · -- Proof that ¬x ∈ D

    done
  done
```

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

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

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

Lean File

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

    done
  done
```

Tactic State in Infoview

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

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

Lean File

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

Tactic State in Infoview

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

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

Lean File

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

Tactic State in Infoview

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

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

Lean File

Tactic State in Infoview

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

No goals

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

To prove a goal of the form `P ↔ Q`:

Prove `P → Q` and `Q → P` separately.

To use a given of the form `P ↔ Q`:

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

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

Let's try these strategies out in an example.

Lean File

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

  done
```

Tactic State in Infoview

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

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

Lean File

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

Tactic State in Infoview

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

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

Lean File

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

    done
  done
```

Tactic State in Infoview

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

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

Lean File

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

    done
  done
```

Tactic State in Infoview

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

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

Lean File

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

    done
  done
```

Tactic State in Infoview

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

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

Lean File

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

    done
  done
```

Tactic State in Infoview

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

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

Lean File

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

    done
  done
```

Tactic State in Infoview

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

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

Lean File

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

Tactic State in Infoview

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

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

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

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

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

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

Proof. Let x be arbitrary. Then

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

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

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

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

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

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

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

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

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

(This theorem says that "iff" has a property called *reflexivity*; we'll discuss reflexivity in Chapter 4.) When variables are declared in the statement of a theorem, it is understood that they can stand for anything of the appropriate type (see Section 3.1 of *HTPI*). Thus, the theorem `Iff.refl` can be thought of as establishing the truth of the statement `∀ (a : Prop), a ↔ a`. In fact, you can get Lean to report the meaning of the theorem in this form with the command `#check @Iff.refl`. What this means is that, in any proof, Lean let's you treat `Iff.refl` as

a proof of the statement ∀ (a : Prop), a ↔ a. Thus, by universal instantiation, for any proposition a, Lean will recognize `Iff.refl` a as a proof of a ↔ a. This is used to justify the first biconditional in the calculational proof.

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

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

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

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

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

which is shorthand for

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

Recall that the curly braces indicate that a, b, and c are implicit arguments, and that Lean groups the logical connectives to the right, which means that it interprets a ∧ b ∧ c as a ∧ (b ∧ c). This is the associative law for "and" (see Section 1.2 of *HTPI*). Since a, b, and c are implicit, Lean will recognize `and_assoc` as a proof of any statement of the form (a ∧ b) ∧ c ↔ a ∧ (b ∧ c), where a, b, and c can be replaced with any propositions. Lean doesn't need to be told what propositions are being used as a, b, and c; it will figure that out for itself. Unfortunately, the second biconditional in the calculational proof is x ∈ A ∧ (x ∈ B ∧ x ∉ C) ↔ (x ∈ A ∧ x ∈ B) ∧ x ∉ C, which has the form a ∧ (b ∧ c) ↔ (a ∧ b) ∧ c, not (a ∧ b) ∧ c ↔ a ∧ (b ∧ c). (Notice that the first of these biconditionals is the same as the second except that the left and right sides have been swapped.) To account for this discrepancy, we use the fact that if h is a proof of any biconditional P ↔ Q, then `h.symm` is a proof of Q ↔ P. Thus `and_assoc.symm`

proves the second biconditional in the calculational proof. (By the way, the *HTPI* proof avoids any mention of the associativity of "and" by simply leaving out parentheses in the conjunction $x \in A \land x \in B \land x \notin C$. As explained in Section 1.2 of *HTPI*, this represents an implicit use of the associativity of "and.")

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

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

which is shorthand for

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

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

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

Exercises

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

  done
```

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

  done
```

3.
```
theorem Exercise_3_3_16 (U : Type) (B : Set U)
    (F : Set (Set U)) : F ⊆ 𝒫 B → U₀F ⊆ B := by

    done
```

4.
```
theorem Exercise_3_3_17 (U : Type) (F G : Set (Set U))
    (h1 : ∀ (A : Set U), A ∈ F → ∀ (B : Set U), B ∈ G → A ⊆ B) :
    U₀F ⊆ ∩₀G := by

    done
```

5.
```
theorem Exercise_3_4_7 (U : Type) (A B : Set U) :
    𝒫 (A ∩ B) = 𝒫 A ∩ 𝒫 B := by

    done
```

6.
```
theorem Exercise_3_4_17 (U : Type) (A : Set U) : A = U₀(𝒫 A) := by

    done
```

7.
```
theorem Exercise_3_4_18a (U : Type) (F G : Set (Set U)) :
    U₀(F ∩ G) ⊆ (U₀F) ∩ (U₀G) := by

    done
```

8.
```
theorem Exercise_3_4_19 (U : Type) (F G : Set (Set U)) :
    (U₀F) ∩ (U₀G) ⊆ U₀(F ∩ G) ↔
        ∀ (A B : Set U), A ∈ F → B ∈ G → A ∩ B ⊆ U₀(F ∩ G) := by

    done
```

3.5. Proofs Involving Disjunctions

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

To use a given of the form `P ∨ Q`:

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

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

Tactic State Before Using Strategy

```
    ⋮
  h : P ∨ Q
  ⊢ goal
```

Tactic State After Using Strategy

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

Notice that the original given `h : P ∨ Q` gets *replaced* by `h : P` in case 1 and `h : Q` in case 2. This is usually what is most convenient, but if you write `by_cases on h with h1`, then the original given `h` will be preserved, and new givens `h1 : P` and `h1 : Q` will be added to cases 1 and 2, respectively. If you want different names for the new givens in the two cases, then use `by_cases on h with h1, h2` to add the new given `h1 : P` in case 1 and `h2 : Q` in case 2.

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

There are two introduction rules that you can use in Lean to prove a goal of the form `P ∨ Q`. If you have `h : P`, then Lean will accept `Or.intro_left h` as a proof of `P ∨ Q`, and if you have `h : Q`, then `Or.intro_right h` is a proof of `P ∨ Q`. There are shortened forms `Or.inl` and `Or.inr` for `Or.intro_left` and `Or.intro_right`, respectively.

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

To prove a goal of the form `P ∨ Q`:

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

Example 3.5.2 from *HTPI* illustrates these strategies:

Lean File

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

  done
```

Tactic State in Infoview

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

The `define` tactic would rewrite the goal as `∀ {a : U}, a ∈ A \ (B \ C) → a ∈ A \ B ∪ C`, which suggests that our next two tactics should be `fix x : U` and `assume h1 : x ∈ A \ (B \ C)`. But as we have seen before, if you know what the result of the `define` tactic is going to be, then there is usually no need to use it. After introducing `x` as an arbitrary element of `A \ (B \ C)`, we write out the definitions of our new given and goal to help guide our next strategy choice:

Lean File

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

Tactic State in Infoview

```
U : Type
A B C : Set U
x : U
h1 : x ∈ A ∧ ¬x ∈ B \ C
⊢ x ∈ A \ B ∨ x ∈ C
```

The goal is now a disjunction, which suggests that proof by cases might be helpful. But what cases should we use? The key is to look at the meaning of the right half of the given `h1`. The meaning of `¬x ∈ B \ C` is `¬(x ∈ B ∧ x ∉ C)`, which, by one of the De Morgan laws, is equivalent to `x ∉ B ∨ x ∈ C`.

Lean File

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

Tactic State in Infoview

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

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

Lean File

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

Tactic State in Infoview

```
case Case_1
U : Type
A B C : Set U
x : U
h1 : x ∈ A ∧ ¬x ∈ B \ C
h2 : ¬x ∈ B
⊢ x ∈ A \ B ∨ x ∈ C
case Case_2
U : Type
A B C : Set U
x : U
h1 : x ∈ A ∧ ¬x ∈ B \ C
h2 : x ∈ C
⊢ x ∈ A \ B ∨ x ∈ C
```

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

Lean File

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

    done
  done
```

Tactic State in Infoview

```
case Case_1.h
U : Type
A B C : Set U
x : U
h1 : x ∈ A ∧ ¬x ∈ B \ C
h2 : ¬x ∈ B
⊢ x ∈ A \ B
```

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

Lean File

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

    done
  done
```

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

Tactic State in Infoview

```
case Case_2
U : Type
A B C : Set U
x : U
h1 : x ∈ A ∧ ¬x ∈ B \ C
h2 : x ∈ C
⊢ x ∈ A \ B ∨ x ∈ C
```

Lean File

Tactic State in Infoview

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

```
No goals
```

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

To prove a goal of the form `P ∨ Q`:

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

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

Tactic State Before Using Strategy

Tactic State After Using Strategy

```
    ⋮
 ⊢ P ∨ Q
```

```
    ⋮
 h : ¬Q
 ⊢ P
```

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

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

Lean File   Tactic State in Infoview

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

  done
```

```
U : Type
A B C : Set U
h1 : A \ B ⊆ C
⊢ A ⊆ B ∪ C
```

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

Lean File   Tactic State in Infoview

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

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

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

Lean File   Tactic State in Infoview

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

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

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

Lean File

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

Tactic State in Infoview

```
No goals
```

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

To use a given of the form `P ∨ Q`:

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

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

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

Lean File

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

  done
```

Tactic State in Infoview

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

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

Lean File   Tactic State in Infoview

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

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

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

Lean File   Tactic State in Infoview

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

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

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

Lean File

Tactic State in Infoview

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

```
No goals
```

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

Lean File

Tactic State in Infoview

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

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

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

Lean File

Tactic State in Infoview

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

```
No goals
```

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

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

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

Lean File

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

    done
  disj_syll h4 h5  --h4: a ∈ C
  show a ∈ C from h4
  done
```

Tactic State in Infoview

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

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

Lean File

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

Tactic State in Infoview

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

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

Lean File   Tactic State in Infoview

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

No goals

The red squiggle has disappeared from the word `done`, indicating that the proof is complete.

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

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

Exercises

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

1.
```
theorem Exercise_3_5_2 (U : Type) (A B C : Set U) :
    (A ∪ B) \ C ⊆ A ∪ (B \ C) := sorry
```

2.
```
theorem Exercise_3_5_5 (U : Type) (A B C : Set U)
    (h1 : A ∩ C ⊆ B ∩ C) (h2 : A ∪ C ⊆ B ∪ C) : A ⊆ B := sorry
```

3.
```
theorem Exercise_3_5_7 (U : Type) (A B C : Set U) :
    A ∪ C ⊆ B ∪ C ↔ A \ C ⊆ B \ C := sorry
```

4.
```
theorem Exercise_3_5_8 (U : Type) (A B : Set U) :
    𝒫 A ∪ 𝒫 B ⊆ 𝒫 (A ∪ B) := sorry
```

5.
```
theorem Exercise_3_5_17b (U : Type) (F : Set (Set U)) (B : Set U) :
    B ∪ (⋂₀ F) =
      ⋂₀ { X : Set U | ∃ (A : Set U), A ∈ F ∧ X = B ∪ A} := sorry
```

6.
```
theorem Exercise_3_5_18 (U : Type) (F G H : Set (Set U))
    (h1 : ∀ (A : Set U), A ∈ F → ∀ (B : Set U), B ∈ G → A ∪ B ∈ H) :
    ⋂₀H ⊆ (⋂₀F) ∪ (⋂₀G) := sorry
```

7.
```
theorem Exercise_3_5_24a (U : Type) (A B C : Set U) :
    (A ∪ B) ∆ C ⊆ (A ∆ C) ∪ (B ∆ C) := sorry
```

3.6. Existence and Uniqueness Proofs

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

Often a goal of the form `∃! (x : U), P x` is proven by using the following strategy. This is a slight rephrasing of the strategy presented in *HTPI*. The rephrasing is based on the fact that for any propositions `A`, `B`, and `C`, `A ∧ B → C` is equivalent to `A → B → C` (you can check this equivalence by making a truth table). The second of these statements is usually easier to work with in Lean than the first one, so we will often rephrase statements that have the form `A ∧ B → C` as `A → B → C`. That's what we have done here, as you will see if you compare the strategy stated below to the one in *HTPI*.

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

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

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

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

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

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

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

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

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

Lean File

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

Tactic State in Infoview

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

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

Lean File

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

    done
  · -- (←)

    done
  done
```

Tactic State in Infoview

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

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

Lean File

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

    done
  done
```

Tactic State in Infoview

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

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

Lean File

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

      done
    done
  · -- (←)

    done
  done
```

Tactic State in Infoview

```
U : Type
B : Set U
x : U
h1 : x ∈ ∅ ∨ x ∈ B
⊢ ¬x ∈ ∅
```

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

Lean File

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

    done
  done
```

Tactic State in Infoview

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

To see how to use the new assumption `h3`, we use the tactic `define at h3`. The definition Lean gives for the statement `x ∈ ∅` is `False`. In other words, Lean knows that, by the definition of

$\varnothing$, the statement $x \in \varnothing$ is false. Since `False` is our goal, this completes the inner proof, and we can return to the main proof.

<table>
<tr><td>Lean File</td><td>Tactic State in Infoview</td></tr>
</table>

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

    done
  done
```

```
case h.mp
U : Type
B : Set U
x : U
h1 : x ∈ ∅ ∨ x ∈ B
h2 : ¬x ∈ ∅
⊢ x ∈ B
```

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

Lean File

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

    done
  done
```

Tactic State in Infoview

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

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

Lean File

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

Tactic State in Infoview

```
No goals
```

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

Lean File

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

  done
```

Tactic State in Infoview

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

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

Lean File   Tactic State in Infoview

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

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

To understand the goal better, we'll write out the definitions of the two sides of the biconditional. We use an extension of the define tactic that allows us to write out the definition of just a part of a given or the goal. The tactic define : x ∈ X ∪ Y will replace x ∈ X ∪ Y with its definition wherever it appears in the goal, and then define : x ∈ Y ∪ X will replace x ∈ Y ∪ X with its definition. (As usual, you can add at to do the replacements in a given rather than the goal.)

Lean File   Tactic State in Infoview

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

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

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

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

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

of the form `a ∨ b ↔ b ∨ a`, where Lean will figure out for itself what `a` and `b` stand for. In particular, `or_comm` will prove our current goal.

Lean File

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

Tactic State in Infoview

```
No goals
```

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

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

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

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

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

Lean File

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

  done
```

Tactic State in Infoview

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

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

Lean File

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

Tactic State in Infoview

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

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

Lean File

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

    done
  done
```

Tactic State in Infoview

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

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

Lean File

Tactic State in Infoview

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

    done
  done
```

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

For the uniqueness proof, we begin by introducing arbitrary sets `C` and `D` and assuming `∀ (B : Set U), C ∪ B = B` and `∀ (B : Set U), D ∪ B = B`, exactly as in the *HTPI* proof.

Lean File

Tactic State in Infoview

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

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

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

Lean File | Tactic State in Infoview

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

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

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

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

```
            = D ∪ C := h4.symm
          _ = C ∪ D := union_comm D C
          _ = D     := h3
      done
    done
```

Since the statement ∃! (x : U), P x asserts both the existence and the uniqueness of an object satisfying the predicate P, we have the following strategy for using a given of this form:

To use a given of the form ∃! (x : U), P x:

Introduce a new variable, say a, into the proof to stand for an object of type U for which P a is true. You may also assert that ∀ (x_1 x_2 : U), P x_1 → P x_2 → x_1 = x2.

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

Lean File

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

  done
```

Tactic State in Infoview

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

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

Lean File

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

Tactic State in Infoview

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

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

Lean File

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

Tactic State in Infoview

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

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

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

Lean File

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

Tactic State in Infoview

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

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

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

```
    obtain (a : U) (h6 : a ∈ A) (h7 : ∀ (y z : U),
      y ∈ A → z ∈ A → y = z)  from h3
    define at h4; define at h5
    have h8 : b = c := h7 b c h4.left h5.left
    rewrite [h8] at h4
    show ∃ (x : U), x ∈ B ∩ C from
      Exists.intro c (And.intro h4.right h5.right)
    done
```

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

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

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

Before ending this section, we return to the question of how you can tell if a theorem you want to use is in Lean's library. In an earlier example, we guessed that the commutative law for "or" might be in Lean's library, and we were then able to use the #check command to confirm it. But there is another technique that we could have used: the tactic library_search. Let's return to our proof of the theorem union_comm, which started like this:

Lean File

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

Tactic State in Infoview

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

Now let's give the library_search tactic a try.

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

```
    define : x ∈ X ∪ Y
    define : x ∈ Y ∪ X
    library_search
    done
```

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

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

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

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

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

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

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

The `library_search` tactic is somewhat unpredictable; sometimes it is able to find the right theorem in the library, and sometimes it isn't. But it is always worth a try. Other tactics that may be helpful are `suggest` and `hint`.

Exercises

1.
```
    theorem Exercise_3_4_15 (U : Type) (B : Set U) (F : Set (Set U)) :
        ⋃₀{X : Set U | ∃ (A : Set U), A ∈ F ∧ X = A \ B}
          ⊆ ⋃₀(F \ 𝒫 B) := sorry
```

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

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

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

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

6.
```
theorem Exercise_3_6_10 (U : Type) (A : Set U)
    (h1 : ∀ (F : Set (Set U)), ⋃₀F = A → A ∈ F) :
    ∃! (x : U), x ∈ A := by
  --Hint:  Start like this:
  let F0 := {X : Set U | X ⊆ A ∧ ∃! (x : U), x ∈ X}
  --Now F0 is in the tactic state, with the definition above
  have h2 : ⋃₀F0 = A := sorry

  done
```

3.7. More Examples of Proofs

It is finally time to discuss proofs involving algebraic reasoning. Lean has types for several different kinds of numbers. `Nat` is the type of natural numbers—that is, the numbers 0, 1, 2, …. `Int` is the type of integers, `Rat` is the type of rational numbers, `Real` is the type of real numbers, and `Complex` is the type of complex numbers. Lean also uses the notation $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{R}$, and $\mathbb{C}$ for these types. To write formulas involving arithmetic operations, you should use the symbols `+` for addition, `-` for subtraction, `*` for multiplication, `/` for division, and `^` for exponentiation. To see what's involved in proving theorems about numbers in Lean, we'll turn to a few examples from earlier in Chapter 3 of *HTPI*.

We begin with Theorem 3.3.7, which concerns divisibility of integers. As in *HTPI*, for integers `x` and `y`, we will write `x | y` to mean that `x` divides `y`, or `y` is divisible by `x`. The formal

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

Lean File	Tactic State in Infoview
``theorem Theorem_3_3_7 : ∀ (a b c : Int), a	b → b	c → a	c := by done```	```⊢ ∀ (a b c : ℤ), a	b → b	c → a	c```

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

Lean File	Tactic State in Infoview
``theorem Theorem_3_3_7 : ∀ (a b c : Int), a	b → b	c → a	c := by fix a : Int; fix b : Int; fix c : Int assume h1 : a	b; assume h2 : b	c define at h1; define at h2; define done```	```a b c : ℤ h1 : ∃ (c : ℤ), b = a * c h2 : ∃ (c_1 : ℤ), c = b * c_1 ⊢ ∃ (c_1 : ℤ), c = a * c_1```

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

Lean File	Tactic State in Infoview
``theorem Theorem_3_3_7 : ∀ (a b c : Int), a	b → b	c → a	c := by fix a : Int; fix b : Int; fix c : Int assume h1 : a	b; assume h2 : b	c define at h1; define at h2; define obtain (m : Int) (h3 : b = a * m) from h1 obtain (n : Int) (h4 : c = b * n) from h2 done```	```a b c : ℤ h1 : ∃ (c : ℤ), b = a * c h2 : ∃ (c_1 : ℤ), c = b * c_1 m : ℤ h3 : b = a * m n : ℤ h4 : c = b * n ⊢ ∃ (c_1 : ℤ), c = a * c_1```

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

Lean File   Tactic State in Infoview

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

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

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

Lean File   Tactic State in Infoview

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

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

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

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

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

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

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

Lean File

Tactic State in Infoview

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

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

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

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

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

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

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

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

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

Proof. Let n be an arbitrary integer.

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

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

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

Lean File

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

Tactic State in Infoview

```
case mp
n : ℤ
⊢ 6 | n → 2 | n ∧ 3 | n
case mpr
n : ℤ
⊢ 2 | n ∧ 3 | n → 6 | n
```

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

Lean File

Tactic State in Infoview

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

    done
  done
```

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

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

Lean File

Tactic State in Infoview

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

      done
    done
  · -- (←)

    done
  done
```

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

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

Lean File

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

      done
    done
  · -- (←)

    done
  done
```

Tactic State in Infoview

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

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

Lean File

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

      done
    done
  · -- (←)

    done
  done
```

Tactic State in Infoview

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

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

Lean File

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

      done
    done
  · -- (←)

    done
  done
```

Tactic State in Infoview

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

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

Lean File

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

    done
  done
```

Tactic State in Infoview

```
case mpr
n : ℤ
⊢ 2 | n ∧ 3 | n → 6 | n
```

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

Lean File

Tactic State in Infoview

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

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

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

Lean File

Tactic State in Infoview

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

```
No goals
```

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

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

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

Don't worry too much about the confusing implicit arguments in the first line; they are implicit, so you won't have to supply them. The second line tells us that we can fill in `mul_sub_left_distrib 6 j k` as a proof of the first equation.

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

Lean File

```
have h6 : 6 * (j - k) = n :=
  calc 6 * (j - k)
      = 6 * j - 6 * k :=
        mul_sub_left_distrib 6 j k
    _ = 3 * (2 * j) - 2 * (3 * k) :=
        by

          done
    _ = 3 * n - 2 * n := sorry
    _ = (3 - 2) * n := sorry
    _ = n := sorry
```

Tactic State in Infoview

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

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

Lean File

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

Tactic State in Infoview

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

To finish off the justification of the second equation, we'll use the theorem `Eq.refl`. The command `#check @Eq.refl` gives the result `@Eq.refl : ∀ {α : Sort u_1} (a : α), a = a`. Ignoring the implicit argument `α`, this should remind you of the theorem `Iff.refl : ∀ (a : Prop), a ↔ a`. Recall that we were able to use `Iff.refl _` to prove not only any statement of the form `a ↔ a`, but also statements of the form `a ↔ a'`, where `a` and `a'` are definitionally equal. Similarly, `Eq.refl _` will prove any equation of the form `a = a'`, where `a` and `a'` are definitionally equal. Since Lean knows that, by definition, `3 * 2 = 6` and `2 * 3 = 6`, the goal

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

Lean File

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

        done
    _ = (3 - 2) * n := sorry
    _ = n := sorry
```

Tactic State in Infoview

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

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

Lean File   Tactic State in Infoview

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

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

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

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

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

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

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

```
example (n : Int) : 1 * n = n := by library_search
```

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

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

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

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

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

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

Exercises

1.
```
theorem Exercise_3_3_18a (a b c : Int)
    (h1 : a | b) (h2 : a | c) : a | (b + c) := sorry
```

2. Complete the following proof by justifying the steps in the calculational proof. Remember that you can use the tactic `demorgan : ...` to apply one of De Morgan's laws to just a part of the goal. You may also find the theorem `and_or_left` useful. (Use `#check` to see what the theorem says.)

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

For the next exercise you will need the following definitions:

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

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

3.
```
theorem Exercise_3_4_10 (x y : Int)
    (h1 : odd x) (h2 : odd y) : even (x - y) := sorry
```

4.
```
theorem Exercise_3_4_27a :
    ∀ (n : Int), 15 | n ↔ 3 | n ∧ 5 | n := sorry
```

4 Relations

4.1. Ordered Pairs and Cartesian Products

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

4.2. Relations

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

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

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

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

The *range* of R is the set

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

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

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

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

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

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

We can write corresponding definitions in Lean as follows:

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

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

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

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

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

Here is the main theorem about these concepts, as stated in *HTPI*:

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

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

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

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

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

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

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

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

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

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

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

Here is the complete proof.

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

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

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

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

```
theorem Theorem_4_2_5_5 {A B C : Type}
    (R : Set (A × B)) (S : Set (B × C)) :
    inv (comp S R) = comp (inv R) (inv S) := by
  apply Set.ext
  fix (c, a) : C × A
  apply Iff.intro
  · -- (→)
    assume h1 : (c, a) ∈ inv (comp S R)
```

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

    done
  done
```

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

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

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

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

Returning to our proof of part 5 of Theorem 4.2.5, recall that after the step `apply Exists.intro b`, the goal is `(c, b) ∈ inv S ∧ (b, a) ∈ inv R`. Rather than using the `define` tactic to write out the definitions of the inverses, we'll use the tactic `rewrite [simp_inv, simp_inv]`. Why do we list `simp_inv` twice in the `rewrite` tactic? When we ask Lean to use the theorem `simp_inv` as a rewriting rule, it figures out that `simp_inv S b c` is a proof of the statement `(c, b) ∈ inv` `S ↔ (b, c) ∈ S`, which can be used to rewrite the left half of the goal. To rewrite the right half, we need a different application of the `simp_inv` theorem, `simp_inv R a b`. So we have to ask Lean to apply the theorem a second time. After the `rewrite` tactic, the goal is `(b, c) ∈ S` `∧ (a, b) ∈ R`, which will follow easily from `h2`.

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

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

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

Exercises

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

2.
```
theorem Exercise_4_2_9b {A B C : Type} (R : Set (A × B))
    (S : Set (B × C)) : Ran R ⊆ Dom S → Dom (comp S R) = Dom R := sorry
```

3.
```
--Fill in the blank to get a correct theorem and then prove the theorem
theorem Exercise_4_2_9c {A B C : Type} (R : Set (A × B))
    (S : Set (B × C)) : ___ → Ran (comp S R) = Ran S := sorry
```

4.
```
theorem Exercise_4_2_12a {A B C : Type}
    (R : Set (A × B)) (S T : Set (B × C)) :
    (comp S R) \ (comp T R) ⊆ comp (S \ T) R := sorry
```

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

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

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

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

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

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

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

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

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

4.3. More About Relations

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

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

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

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

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

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

The rest of Chapter 4 of *HTPI* focuses on relations from a set to itself; in Lean, the corresponding idea is a relation from a type to itself. If `A` is any type and `R` has type `Rel A A`, then we will say that `R` is a *binary relation on* `A`. The notation `BinRel A` denotes the type of binary relations on `A`. In other words, `BinRel A` is just an abbreviation for `Rel A A`. If `R` is a binary relation on `A`, then we say that `R` is *reflexive* if for every `x` of type `A`, `R x x` holds. It is *symmetric* if for all `x` and `y` of type `A`, if `R x y` then `R y x`. And it is *transitive* if for all `x`, `y`, and `z` of type

A, if R x y and R y z then R x z. Of course, we can tell Lean about these definitions, which correspond to Definition 4.3.2 in *HTPI*:

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

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

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

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

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

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

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

```
theorem Theorem_4_3_4_2 {A : Type} (R : BinRel A) :
    symmetric R ↔ extension R = inv (extension R) := by
  apply Iff.intro
  · -- (→)
    assume h1 : symmetric R
    define at h1                --h1: ∀ (x y : A), R x y → R y x
    apply Set.ext
    fix (a, b) : A × A
    show (a, b) ∈ extension R ↔ (a, b) ∈ inv (extension R) from
      calc (a, b) ∈ extension R
          ↔ R a b := by rfl
        _ ↔ R b a := Iff.intro (h1 a b) (h1 b a)
```

```
          _ ↔ (a, b) ∈ inv (extension R) := by rfl
      done
  · -- (←)
      assume h1 : extension R = inv (extension R)
      define                    --Goal: ∀ (x y : A), R x y → R y x
      fix a : A; fix b : A
      assume h2 : R a b         --Goal: R b a
      rewrite [←simp_ext R, h1, simp_inv, simp_ext] at h2
      show R b a from h2
      done
  done
```

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

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

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

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

This definition says that if `A` is a type, `a` has type `A`, and `X` has type `Set A`, then `elementhood A a X` is the proposition `a ∈ X`. Thus, if `elementhood A` is followed by objects of type `A` and `Set A`, the result is a proposition, so `elementhood A` is functioning as a relation from `A` to `Set A`. For example, `elementhood Int` is a relation from integers to sets of integers, and `elementhood Int 6 { n : Int | ∃ (k : Int), n = 2 * k }` is the (true) statement that `6` is an element of the set of even integers. (You are asked to prove it in the exercises.)

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

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

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

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

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

Exercises

1.
```
example :
    elementhood Int 6 { n : Int | ∃ (k : Int), n = 2 * k } := sorry
```

2.
```
theorem Theorem_4_3_4_1 {A : Type} (R : BinRel A) :
    reflexive R ↔ { (x, y) : A × A | x = y } ⊆ extension R := sorry
```

3.
```
theorem Theorem_4_3_4_3 {A : Type} (R : BinRel A) :
    transitive R ↔
        comp (extension R) (extension R) ⊆ extension R := sorry
```

4.
```
theorem Exercise_4_3_12a {A : Type} (R : BinRel A) (h1 : reflexive R) :
    reflexive (relFromExt (inv (extension R))) := sorry
```

5.
```
theorem Exercise_4_3_12c {A : Type} (R : BinRel A) (h1 : transitive R) :
    transitive (relFromExt (inv (extension R))) := sorry
```

6.
```
theorem Exercise_4_3_18 {A : Type}
    (R S : BinRel A) (h1 : transitive R) (h2 : transitive S)
    (h3 : comp (extension S) (extension R) ⊆
        comp (extension R) (extension S)) :
    transitive (relFromExt (comp (extension R) (extension S))) := sorry
```

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

7.
```
--You might not be able to complete this proof
theorem Exercise_4_3_13b {A : Type}
    (R1 R2 : BinRel A) (h1 : symmetric R1) (h2 : symmetric R2) :
    symmetric (relFromExt ((extension R1) ∪ (extension R2))) := sorry
```

8.
```
--You might not be able to complete this proof
theorem Exercise_4_3_13c {A : Type}
    (R1 R2 : BinRel A) (h1 : transitive R1) (h2 : transitive R2) :
    transitive (relFromExt ((extension R1) ∪ (extension R2))) := sorry
```

4.4. Ordering Relations

Section 4.4 of *HTPI* begins by defining several new concepts about binary relations. Here are the definitions, written in Lean:

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

def partial_order {A : Type} (R : BinRel A) : Prop :=
    reflexive R ∧ transitive R ∧ antisymmetric R

def total_order {A : Type} (R : BinRel A) : Prop :=
    partial_order R ∧ ∀ (x y : A), R x y ∨ R y x
```

These definitions say that if `R` is a binary relation on `A`, then `R` is *antisymmetric* if `R x y` and `R y x` cannot both be true unless `x = y`. `R` is a *partial order on* `A`—or just a *partial order*, if `A` is clear from context—if it is reflexive, transitive, and antisymmetric. And `R` is a *total order on* `A` if it is a partial order and also, for any `x` and `y` of type `A`, either `R x y` or `R y x`. Note that, since Lean groups the connective `∧` to the right, `partial_order R` means `reflexive R ∧ (transitive R ∧ antisymmetric R)`, and therefore if `h` is a proof of `partial_order R`, then `h.left` is a proof of `reflexive R`, `h.right.left` is a proof of `transitive R`, and `h.right.right` is a proof of `antisymmetric R`.

Example 4.4.3 in *HTPI* gives several examples of partial orders and total orders. We'll give one of those examples here. For any type `A`, we define `sub A` to be the subset relation on sets of objects of type `A`:

```
def sub (A : Type) (X Y : Set A) : Prop := X ⊆ Y
```

According to this definition, `sub A` is a binary relation on `Set A`, and for any two sets `X` and `Y` of type `Set A`, `sub A X Y` is the proposition `X ⊆ Y`. We will leave it as an exercise for you to prove that `sub A` is a partial order on the type `Set A`.

Notice that `X ⊆ Y` could be thought of as expressing a sense in which `Y` is "at least as large as" `X`. Often, if `R` is a partial order on `A` and `a` and `b` have type `A`, then `R a b` can be thought of as meaning that `b` is in some sense "at least as large as" `a`. Many of the concepts we study for partial and total orders are motivated by this interpretation of `R`.

For example, if `R` is a partial order on `A`, `B` has type `Set A`, and `b` has type `A`, then we say that `b` is an `R`-*smallest element* of `B` if it is an element of `B`, and every element of `B` is at least as large as `b`, according to this interpretation of the ordering `R`. We say that `b` is an `R`-*minimal element* of `B` if it is an element of `B`, and there is no other element of `B` that is smaller than `b`, according to the ordering `R`. We can state these precisely as definitions in Lean:

```
def smallestElt {A : Type} (R : BinRel A) (b : A) (B : Set A) : Prop :=
    b ∈ B ∧ ∀ x ∈ B, R b x

def minimalElt {A : Type} (R : BinRel A) (b : A) (B : Set A) : Prop :=
    b ∈ B ∧ ¬∃ x ∈ B, R x b ∧ x ≠ b
```

Notice that, as in *HTPI*, in Lean we can write `∀ x ∈ B, P x` as an abbreviation for `∀ (x : A), x ∈ B → P x`, and `∃ x ∈ B, P x` as an abbreviation for `∃ (x : A), x ∈ B ∧ P x`. According to these definitions, `smallestElt R b B` is the proposition that `b` is an `R`-smallest element of `B`, and `minimalElt R b B` means that `b` is an `R`-minimal element of `B`.

Theorem 4.4.6 in *HTPI* asserts three statements about these concepts. We'll prove the second and third, and leave the first as an exercise for you. The first statement in Theorem 4.4.6 says that if `B` has an `R`-smallest element, then that `R`-smallest element is unique. Thus, we can talk about *the* `R`-smallest element of `B` rather than *an* `R`-smallest element. The second says that if `b` is the `R`-smallest element of `B`, then it is also an `R`-minimal element, and it is the only `R`-minimal element. Here is how you might start the proof:

```
theorem Theorem_4_4_6_2 {A : Type} (R : BinRel A) (B : Set A) (b : A)
    (h1 : partial_order R) (h2 : smallestElt R b B) :
    minimalElt R b B ∧ ∀ (c : A), minimalElt R c B → b = c := by
  define at h1     --h1: reflexive R ∧ transitive R ∧ antisymmetric R
  define at h2     --h2: b ∈ B ∧ ∀ (x : A), x ∈ B → R b x
  apply And.intro
  · -- Proof that b is minimal
    define          --Goal: b ∈ B ∧ ¬∃ (x : A), x ∈ B ∧ R x b ∧ x ≠ b
    apply And.intro h2.left
    quant_neg       --Goal: ∀ (x : A), ¬(x ∈ B ∧ R x b ∧ x ≠ b)
```

```
      demorgan : ¬(x ∈ B ∧ R x b ∧ x ≠ b)
      done
  · -- Proof that b is only minimal element

      done
    done
```

When the goal is ∀ (x : A), ¬(x ∈ B ∧ R x b ∧ x ≠ b), it is tempting to apply the demorgan tactic to ¬(x ∈ B ∧ R x b ∧ x ≠ b), but unfortunately this generates an error in Lean: unknown identifier 'x'. The problem is that x is not defined in the tactic state, so without the quantifier ∀ (x : A) in front of it, ¬(x ∈ B ∧ R x b ∧ x ≠ b) doesn't mean anything to Lean. The solution to the problem is to deal with the universal quantifier first by introducing an arbitrary x of type A. Once x has been introduced, we can apply the demorgan tactic.

```
theorem Theorem_4_4_6_2 {A : Type} (R : BinRel A) (B : Set A) (b : A)
    (h1 : partial_order R) (h2 : smallestElt R b B) :
    minimalElt R b B ∧ ∀ (c : A), minimalElt R c B → b = c := by
  define at h1     --h1: reflexive R ∧ transitive R ∧ antisymmetric R
  define at h2     --h2: b ∈ B ∧ ∀ (x : A), x ∈ B → R b x
  apply And.intro
  · -- Proof that b is minimal
    define           --Goal: b ∈ B ∧ ¬∃ (x : A), x ∈ B ∧ R x b ∧ x ≠ b
    apply And.intro h2.left
    quant_neg        --Goal: ∀ (x : A), ¬(x ∈ B ∧ R x b ∧ x ≠ b)
    fix x : A
    demorgan         --Goal: ¬x ∈ B ∨ ¬(R x b ∧ x ≠ b)
    or_right with h3 --h3: x ∈ B; Goal: ¬(R x b ∧ x ≠ b)
    demorgan         --Goal: ¬R x b ∨ x = b
    or_right with h4 --h4: R x b; Goal: x = b
    have h5 : R b x := h2.right x h3
    have h6 : antisymmetric R := h1.right.right
    define at h6     --h6: ∀ (x y : A), R x y → R y x → x = y
    show x = b from h6 x b h4 h5
    done
  · -- Proof that b is only minimal element
    fix c : A
    assume h3 : minimalElt R c B
    define at h3     --h3: c ∈ B ∧ ¬∃ (x : A), x ∈ B ∧ R x c ∧ x ≠ c
    contradict h3.right with h4
                     --h4: ¬b = c; Goal: ∃ (x : A), x ∈ B ∧ R x c ∧ x ≠ c
    have h5 : R b c := h2.right c h3.left
    show ∃ (x : A), x ∈ B ∧ R x c ∧ x ≠ c from
```

```
      Exists.intro b (And.intro h2.left (And.intro h5 h4))
    done
  done
```

Finally, the third statement in Theorem 4.4.6 says that if `R` is a total order, then any `R`-minimal element of a set `B` must be the `R`-smallest element of `B`. The beginning of the proof is straightforward:

```
theorem Theorem_4_4_6_3 {A : Type} (R : BinRel A) (B : Set A) (b : A)
    (h1 : total_order R) (h2 : minimalElt R b B) : smallestElt R b B := by
  define at h1         --h1: partial_order R ∧ ∀ (x y : A), R x y ∨ R y x
  define at h2         --h2: b ∈ B ∧ ¬∃ (x : A), x ∈ B ∧ R x b ∧ x ≠ b
  define               --Goal: b ∈ B ∧ ∀ (x : A), x ∈ B → R b x
  apply And.intro h2.left  --Goal: ∀ (x : A), x ∈ B → R b x
  fix x : A
  assume h3 : x ∈ B        --Goal: R b x
  done
```

Surprisingly, at this point it is difficult to find a way to reach the goal `R b x`. See *HTPI* for an explanation of why it turns out to be helpful to split the proof into two cases, depending on whether or not `x = b`. Of course, we use the `by_cases` tactic for this.

```
theorem Theorem_4_4_6_3 {A : Type} (R : BinRel A) (B : Set A) (b : A)
    (h1 : total_order R) (h2 : minimalElt R b B) : smallestElt R b B := by
  define at h1         --h1: partial_order R ∧ ∀ (x y : A), R x y ∨ R y x
  define at h2         --h2: b ∈ B ∧ ¬∃ (x : A), x ∈ B ∧ R x b ∧ x ≠ b
  define               --Goal: b ∈ B ∧ ∀ (x : A), x ∈ B → R b x
  apply And.intro h2.left  --Goal: ∀ (x : A), x ∈ B → R b x
  fix x : A
  assume h3 : x ∈ B        --Goal: R b x
  by_cases h4 : x = b
  · -- Case 1. h4: x = b
    rewrite [h4]           --Goal: R b b
    have h5 : partial_order R := h1.left
    define at h5
    have h6 : reflexive R := h5.left
    define at h6
    show R b b from h6 b
    done
  · -- Case 2. h4: x ≠ b
    have h5 : ∀ (x y : A), R x y ∨ R y x := h1.right
    have h6 : R x b ∨ R b x := h5 x b
```

```
      have h7 : ¬R x b := by
        contradict h2.right with h8
        show ∃ (x : A), x ∈ B ∧ R x b ∧ x ≠ b from
          Exists.intro x (And.intro h3 (And.intro h8 h4))
        done
      disj_syll h6 h7
      show R b x from h6
      done
    done
```

Imitating the definitions above, you should be able to formulate definitions of `R`-largest and `R`-maximal elements. Section 4.4 of *HTPI* defines four more terms: upper bound, lower bound, least upper bound, and greatest lower bound. We will discuss upper bounds and least upper bounds, and leave lower bounds and greatest lower bounds for you to figure out on your own.

If `R` is a partial order on `A`, `B` has type `Set A`, and `a` has type `A`, then `a` is called an *upper bound* for `B` if it is at least as large as every element of `B`. If it is the smallest element of the set of upper bounds, then it is called the *least upper bound* of `B`. The phrase "least upper bound" is often abbreviated "lub". Here are these definitions, written in Lean:

```
  def upperBd {A : Type} (R : BinRel A) (a : A) (B : Set A) : Prop :=
      ∀ x ∈ B, R x a

  def lub {A : Type} (R : BinRel A) (a : A) (B : Set A) : Prop :=
      smallestElt R a { c : A | upperBd R c B }
```

As usual, we will let you consult *HTPI* for examples of these concepts. But we will mention one example: If `A` is a type and `F` has type `Set (Set A)`—that is, `F` is a set whose elements are sets of objects of type `A`—then the least upper bound of `F`, with respect to the partial order `sub A`, is `⋃₀F`. We leave the proof of this fact as an exercise.

Exercises

1.
```
    theorem Example_4_4_3_1 {A : Type} : partial_order (sub A) := sorry
```

2.
```
    theorem Theorem_4_4_6_1 {A : Type} (R : BinRel A) (B : Set A) (b : A)
        (h1 : partial_order R) (h2 : smallestElt R b B) :
        ∀ (c : A), smallestElt R c B → b = c := sorry
```

3.
```
--If F is a set of sets, then U₀F is the lub of F in the subset ordering
theorem Theorem_4_4_11 {A : Type} (F : Set (Set A)) :
    lub (sub A) (U₀F) F := sorry
```

4.
```
theorem Exercise_4_4_8 {A B : Type} (R : BinRel A) (S : BinRel B)
    (T : BinRel (A × B)) (h1 : partial_order R) (h2 : partial_order S)
    (h3 : ∀ (a a' : A) (b b' : B),
        T (a, b) (a', b') ↔ R a a' ∧ S b b') :
    partial_order T := sorry
```

5.
```
theorem Exercise_4_4_9_part {A B : Type} (R : BinRel A) (S : BinRel B)
    (L : BinRel (A × B)) (h1 : total_order R) (h2 : total_order S)
    (h3 : ∀ (a a' : A) (b b' : B),
        L (a, b) (a', b') ↔ R a a' ∧ (a = a' → S b b')) :
    ∀ (a a' : A) (b b' : B),
        L (a, b) (a', b') ∨ L (a', b') (a, b) := sorry
```

6.
```
theorem Exercise_4_4_15a {A : Type}
    (R1 R2 : BinRel A) (B : Set A) (b : A)
    (h1 : partial_order R1) (h2 : partial_order R2)
    (h3 : extension R1 ⊆ extension R2) :
    smallestElt R1 b B → smallestElt R2 b B := sorry
```

7.
```
theorem Exercise_4_4_15b {A : Type}
    (R1 R2 : BinRel A) (B : Set A) (b : A)
    (h1 : partial_order R1) (h2 : partial_order R2)
    (h3 : extension R1 ⊆ extension R2) :
    minimalElt R2 b B → minimalElt R1 b B := sorry
```

8.
```
theorem Exercise_4_4_18a {A : Type}
    (R : BinRel A) (B1 B2 : Set A) (h1 : partial_order R)
    (h2 : ∀ x ∈ B1, ∃ y ∈ B2, R x y) (h3 : ∀ x ∈ B2, ∃ y ∈ B1, R x y) :
    ∀ (x : A), upperBd R x B1 ↔ upperBd R x B2 := sorry
```

9.
```
theorem Exercise_4_4_22 {A : Type}
    (R : BinRel A) (B1 B2 : Set A) (x1 x2 : A)
    (h1 : lub R x1 B1) (h2 : lub R x2 B2) :
    B1 ⊆ B2 → R x1 x2 := sorry
```

10.
```
theorem Exercise_4_4_24 (A : Type) (R : Set (A × A)) :
    smallestElt (sub (A × A)) (R ∪ (inv R))
    { T : Set (A × A) | R ⊆ T ∧ symmetric (relFromExt T) } := sorry
```

4.5. Equivalence Relations

Chapter 4 of *HTPI* concludes with the study of one more important combination of properties that a relation might have. A binary relation R on a set A is called an *equivalence relation* if it is reflexive, symmetric, and transitive. If $x \in A$, then the *equivalence class* of x with respect to R is the set of all $y \in A$ such that yRx. In *HTPI*, this equivalence class is denoted $[x]_R$, so we have

$$[x]_R = \{y \in A \mid yRx\}.$$

The set whose elements are all of these equivalence classes is called A *mod* R. It is written A/R, so

$$A/R = \{[x]_R \mid x \in A\}.$$

Note that A/R is a set whose elements are sets: for each $x \in A$, $[x]_R$ is a subset of A, and $[x]_R \in A/R$.

To define these concepts in Lean, we write:

```
def equiv_rel {A : Type} (R : BinRel A) : Prop :=
    reflexive R ∧ symmetric R ∧ transitive R

def equivClass {A : Type} (R : BinRel A) (x : A) : Set A :=
    { y : A | R y x }

def mod (A : Type) (R : BinRel A) : Set (Set A) :=
    { equivClass R x | x : A }
```

Thus, `equiv_rel R` is the proposition that `R` is an equivalence relation, `equivClass R x` is the equivalence class of `x` with respect to `R`, and `mod A R` is A mod R. Note that `equivClass R x` has type `Set A`, while `mod A R` has type `Set (Set A)`. The definition of `mod A R` is shorthand for `{ X : Set A | ∃ (x : A), equivClass R x = X }`.

HTPI gives several examples of equivalence relations, and these examples illustrate that equivalence classes always have certain properties. The most important of these are that each equivalence class is a nonempty set, the equivalence classes do not overlap, and their union is all of `A`. We say that the equivalence classes form a *partition* of `A`. To state and prove these properties in Lean we will need some definitions. We start with these:

```
def empty {A : Type} (X : Set A) : Prop := ¬∃ (x : A), x ∈ X

def pairwise_disjoint {A : Type} (F : Set (Set A)) : Prop :=
    ∀ X ∈ F, ∀ Y ∈ F, X ≠ Y → empty (X ∩ Y)
```

To say that a set X is empty, we could write X = ∅, but it is more convenient to have a statement that says more explicitly what it means for a set to be empty. Thus, we have defined empty X to be the proposition saying that X has no elements. If F has type Set (Set A), then pairwise_disjoint F is the proposition that no two distinct elements of F have any element in common—in other words, the elements of F do not overlap. We can now give the precise definition of a partition:

```
def partition {A : Type} (F : Set (Set A)) : Prop :=
    (∀ (x : A), x ∈ U₀F) ∧ pairwise_disjoint F ∧ ∀ X ∈ F, ¬empty X
```

The main theorem about equivalence relations in *HTPI* is Theorem 4.5.4, which says that mod A R is a partition of A. The proof of this theorem is hard enough that *HTPI* proves two facts about equivalence classes first. A fact that is proven just for the purpose of using it to prove something else is often called a *lemma*. We can use this term in Lean as well. Here is the first part of Lemma 4.5.5 from *HTPI*

```
lemma Lemma_4_5_5_1 {A : Type} (R : BinRel A) (h : equiv_rel R) :
    ∀ (x : A), x ∈ equivClass R x := by
  fix x : A
  define            --Goal: R x x
  define at h       --h: reflexive R ∧ symmetric R ∧ transitive R
  have href : reflexive R := h.left
  show R x x from href x
  done
```

The command #check @Lemma_4_5_5_1 produces the result

```
@Lemma_4_5_5_1 : ∀ {A : Type} (R : BinRel A),
                 equiv_rel R → ∀ (x : A), x ∈ equivClass R x
```

Thus, if we have R : BinRel A, h : equiv_rel R, and x : A, then Lemma_4_5_5_1 R h x is a proof of x ∈ equivClass R x. We will use this at the end of the proof of our next lemma:

```
lemma Lemma_4_5_5_2 {A : Type} (R : BinRel A) (h : equiv_rel R) :
    ∀ (x y : A), y ∈ equivClass R x ↔
      equivClass R y = equivClass R x := by
  have hsymm : symmetric R := h.right.left
```

```
  have htrans : transitive R := h.right.right
  fix x : A; fix y : A
  apply Iff.intro
  · -- (→)
    assume h2 : y ∈ equivClass R x --Goal: equivClass R y = equivClass R x
    define at h2                    --h2: R y x
    apply Set.ext
    fix z : A
    apply Iff.intro
    · -- Proof that z ∈ equivClass R y → z ∈ equivClass R x
      assume h3 : z ∈ equivClass R y
      define                        --Goal: R z x
      define at h3                  --h3: R z y
      show R z x from htrans z y x h3 h2
      done
    · -- Proof that z ∈ equivClass R x → z ∈ equivClass R y
      assume h3 : z ∈ equivClass R x
      define                        --Goal: R z y
      define at h3                  --h3: R z x
      have h4 : R x y := hsymm y x h2
      show R z y from htrans z x y h3 h4
      done
    done
  · -- (←)
    assume h2 : equivClass R y = equivClass R x --Goal: y ∈ equivClass R x
    rewrite [←h2]                   --Goal: y ∈ equivClass R y
    show y ∈ equivClass R y from Lemma_4_5_5_1 R h y
    done
  done
```

The definition of "partition" has three parts, so to prove Theorem 4.5.4 we will have to prove three statements. It will make the proof easier to read if we prove the three statements separately.

```
lemma Theorem_4_5_4_part_1 {A : Type} (R : BinRel A) (h : equiv_rel R) :
    ∀ (x : A), x ∈ U₀(mod A R) := by
  fix x : A
  define        --Goal: ∃ (t : Set A), t ∈ mod A R ∧ x ∈ t
  apply Exists.intro (equivClass R x)
  apply And.intro _ (Lemma_4_5_5_1 R h x)
                --Goal: equivClass R x ∈ mod A R
  define        --Goal: ∃ (x_1 : A), equivClass R x_1 = equivClass R x
```

```
    apply Exists.intro x
    rfl

  lemma Theorem_4_5_4_part_2 {A : Type} (R : BinRel A) (h : equiv_rel R) :
      pairwise_disjoint (mod A R) := by
    define
    fix X : Set A
    assume h2 : X ∈ mod A R
    fix Y : Set A
    assume h3 : Y ∈ mod A R          --Goal: X ≠ Y → empty (X ∩ Y)
    define at h2; define at h3
    obtain (x : A) (h4 : equivClass R x = X) from h2
    obtain (y : A) (h5 : equivClass R y = Y) from h3
    contrapos
    assume h6 : ∃ (x : A), x ∈ X ∩ Y  --Goal: X = Y
    obtain (z : A) (h7 : z ∈ X ∩ Y) from h6
    define at h7
    rewrite [←h4, ←h5] at h7 --h7: z ∈ equivClass R x ∧ z ∈ equivClass R y
    have h8 : equivClass R z = equivClass R x :=
      (Lemma_4_5_5_2 R h x z).ltr h7.left
    have h9 : equivClass R z = equivClass R y :=
      (Lemma_4_5_5_2 R h y z).ltr h7.right
    show X = Y from
      calc X
          = equivClass R x := h4.symm
        _ = equivClass R z := h8.symm
        _ = equivClass R y := h9
        _ = Y              := h5
    done

  lemma Theorem_4_5_4_part_3 {A : Type} (R : BinRel A) (h : equiv_rel R) :
      ∀ X ∈ mod A R, ¬empty X := by
    fix X : Set A
    assume h2 : X ∈ mod A R  --Goal: ¬empty X
    define; double_neg        --Goal: ∃ (x : A), x ∈ X
    define at h2              --h2: ∃ (x : A), equivClass R x = X
    obtain (x : A) (h3 : equivClass R x = X) from h2
    rewrite [←h3]
    show ∃ (x_1 : A), x_1 ∈ equivClass R x from
      Exists.intro x (Lemma_4_5_5_1 R h x)
    done
```

Its easy now to put everything together to prove Theorem 4.5.4.

```
theorem Theorem_4_5_4 {A : Type} (R : BinRel A) (h : equiv_rel R) :
    partition (mod A R) := And.intro (Theorem_4_5_4_part_1 R h)
      (And.intro (Theorem_4_5_4_part_2 R h) (Theorem_4_5_4_part_3 R h))
```

Theorem 4.5.4 shows that an equivalence relation on `A` determines a partition of `A`, namely `mod A R`. Our next project will be to prove Theorem 4.5.6 in *HTPI*, which says that every partition of `A` arises in this way; that is, every partition is `mod A R` for some equivalence relation `R`. To prove this, we must show how to use a partition `F` to define an equivalence relation `R` for which `mod A R = F`. The proof in *HTPI* defines the required equivalence relation `R` as a set of ordered pairs, but in Lean we will need to define it instead as a binary relation on `A`. Translating *HTPI*'s set-theoretic definition into Lean's notation for binary relations leads to the following definition:

```
def EqRelFromPart {A : Type} (F : Set (Set A)) (x y : A) : Prop :=
    ∃ X ∈ F, x ∈ X ∧ y ∈ X
```

In other words, `EqRelFromPart F` is the binary relation on `A` that is true of any two objects `x` and `y` of type `A` if and only if `x` and `y` belong to the same set in `F`. Our plan now is to show that if `F` is a partition of `A`, then `EqRelFromPart F` is an equivalence relation on `A`, and `mod A (EqRelFromPart F) = F`.

Once again, *HTPI* breaks the proof up by proving some lemmas first, and we will find it convenient to break the proof into even smaller pieces. We will leave the proofs of most of these lemmas as exercises for you.

```
lemma overlap_implies_equal {A : Type}
    (F : Set (Set A)) (h : partition F) :
    ∀ X ∈ F, ∀ Y ∈ F, ∀ (x : A), x ∈ X → x ∈ Y → X = Y := sorry

lemma Lemma_4_5_7_ref {A : Type} (F : Set (Set A)) (h : partition F):
    reflexive (EqRelFromPart F) := sorry

lemma Lemma_4_5_7_symm {A : Type} (F : Set (Set A)) (h : partition F):
    symmetric (EqRelFromPart F) := sorry

lemma Lemma_4_5_7_trans {A : Type} (F : Set (Set A)) (h : partition F):
    transitive (EqRelFromPart F) := sorry
```

It is now easy to put these pieces together to prove Lemma 4.5.7 in *HTPI*:

```
lemma Lemma_4_5_7 {A : Type} (F : Set (Set A)) (h : partition F) :
    equiv_rel (EqRelFromPart F) := And.intro (Lemma_4_5_7_ref F h)
      (And.intro (Lemma_4_5_7_symm F h) (Lemma_4_5_7_trans F h))
```

We need one more lemma before we can prove Theorem 4.5.6:

```
lemma Lemma_4_5_8 {A : Type} (F : Set (Set A)) (h : partition F) :
    ∀ X ∈ F, ∀ x ∈ X, equivClass (EqRelFromPart F) x = X := sorry
```

We are finally now ready to address Theorem 4.5.6. Here is the statement of the theorem:

```
theorem Theorem_4_5_6 {A : Type} (F : Set (Set A)) (h: partition F) :
    ∃ (R : BinRel A), equiv_rel R ∧ mod A R = F
```

Of course, the relation `R` that we will use to prove the theorem is `EqRelFromPart F`, so we could start the proof with the tactic `apply Exists.intro (EqRelFromPart F)`. But this means that the rest of the proof will involve many statements about the relation `EqRelFromPart F`. When a complicated object appears multiple times in a proof, it can make the proof a little easier to read if we give that object a name. We can do that by using a new tactic. The tactic `let R : BinRel A := EqRelFromPart F` introduces the new variable `R` into the tactic state. The variable `R` has type `BinRel A`, and it is definitionally equal to `EqRelFromPart F`. That means that, when necessary, Lean will fill in this definition of `R`. For example, one of our first steps will be to apply `Lemma_4_5_7` to `F` and `h`. The conclusion of that lemma is `equiv_rel (EqRelFromPart F)`, but Lean will recognize this as meaning the same thing as `equiv_rel R`. Here is the proof of the theorem:

```
theorem Theorem_4_5_6 {A : Type} (F : Set (Set A)) (h: partition F) :
    ∃ (R : BinRel A), equiv_rel R ∧ mod A R = F := by
  let R : BinRel A := EqRelFromPart F
  apply Exists.intro R                 --Goal: equiv_rel R ∧ mod A R = F
  apply And.intro (Lemma_4_5_7 F h) --Goal: mod A R = F
  apply Set.ext
  fix X : Set A                        --Goal:  X ∈ mod A R ↔ X ∈ F
  apply Iff.intro
  · -- (→)
    assume h2 : X ∈ mod A R           --Goal: X ∈ F
    define at h2                       --h2: ∃ (x : A), equivClass R x = X
    obtain (x : A) (h3 : equivClass R x = X) from h2
    have h4 : x ∈ U₀F := h.left x
    define at h4
    obtain (Y : Set A) (h5 : Y ∈ F ∧ x ∈ Y) from h4
    have h6 : equivClass R x = Y :=
```

```
      Lemma_4_5_8 F h Y h5.left x h5.right
    rewrite [←h3, h6]
    show Y ∈ F from h5.left
    done
  · -- (←)
    assume h2 : X ∈ F                  --Goal: X ∈ mod A R
    have h3 : ¬empty X := h.right.right X h2
    define at h3; double_neg at h3    --h3: ∃ (x : A), x ∈ X
    obtain (x : A) (h4 : x ∈ X) from h3
    define                            --Goal: ∃ (x : A), equivClass R x = X
    show ∃ (x : A), equivClass R x = X from
      Exists.intro x (Lemma_4_5_8 F h X h2 x h4)
    done
  done
```

Exercises

1.
```
lemma overlap_implies_equal {A : Type}
    (F : Set (Set A)) (h : partition F) :
    ∀ X ∈ F, ∀ Y ∈ F, ∀ (x : A), x ∈ X → x ∈ Y → X = Y := sorry
```

2.
```
lemma Lemma_4_5_7_ref {A : Type} (F : Set (Set A)) (h : partition F):
    reflexive (EqRelFromPart F) := sorry
```

3.
```
lemma Lemma_4_5_7_symm {A : Type} (F : Set (Set A)) (h : partition F):
    symmetric (EqRelFromPart F) := sorry
```

4.
```
lemma Lemma_4_5_7_trans {A : Type} (F : Set (Set A)) (h : partition F):
    transitive (EqRelFromPart F) := sorry
```

5.
```
lemma Lemma_4_5_8 {A : Type} (F : Set (Set A)) (h : partition F) :
    ∀ X ∈ F, ∀ x ∈ X, equivClass (EqRelFromPart F) x = X := sorry
```

6.
```
lemma elt_mod_equiv_class_of_elt
    {A : Type} (R : BinRel A) (h : equiv_rel R) :
    ∀ X ∈ mod A R, ∀ x ∈ X, equivClass R x = X := sorry
```

The next three exercises use the following definitions:

```
def dot {A : Type} (F G : Set (Set A)) : Set (Set A) :=
    { Z : Set A | ¬empty Z ∧ ∃ X ∈ F, ∃ Y ∈ G, Z = X ∩ Y }

def conj {A : Type} (R S : BinRel A) (x y : A) : Prop :=
    R x y ∧ S x y
```

7.
```
theorem Exercise_4_5_20a {A : Type} (R S : BinRel A)
    (h1 : equiv_rel R) (h2 : equiv_rel S) :
    equiv_rel (conj R S) := sorry
```

8.
```
theorem Exercise_4_5_20b {A : Type} (R S : BinRel A)
    (h1 : equiv_rel R) (h2 : equiv_rel S) :
    ∀ (x : A), equivClass (conj R S) x =
      equivClass R x ∩ equivClass S x := sorry
```

9.
```
theorem Exercise_4_5_20c {A : Type} (R S : BinRel A)
    (h1 : equiv_rel R) (h2 : equiv_rel S) :
    mod A (conj R S) = dot (mod A R) (mod A S) := sorry
```

The next exercise uses the following definition:

```
def equiv_mod (m x y : Int) : Prop := m | (x - y)
```

10.
```
theorem Theorem_4_5_10 : ∀ (m : Int), equiv_rel (equiv_mod m) := sorry
```

5 Functions

5.1. Functions

The first definition in Chapter 5 of *HTPI* says that if $F \subseteq A \times B$, then F is called a *function* from A to B if for every $a \in A$ there is exactly one $b \in B$ such that $(a, b) \in F$. The notation $F : A \to B$ means that F is a function from A to B. If F is a function from A to B and $a \in A$, then *HTPI* introduces the notation $F(a)$ for the unique $b \in B$ such that $(a, b) \in F$. Thus, if $F : A \to B$, $a \in A$, and $b \in B$, then $F(a) = b$ means the same thing as $(a, b) \in F$. We sometimes think of F as representing an operation that can be applied to an element a of A to produce a corresponding element $F(a)$ of B, and we call $F(a)$ the *value of F at a*, or the *result of applying F to a*.

This might remind you of the situation we faced in Chapter 4. If $R \subseteq A \times B$, $a \in A$, and $b \in B$, then Chapter 4 of *HTPI* uses the notation aRb to mean the same thing as $(a, b) \in R$. But in Lean, we found it necessary to change this notation. Instead of using *HTPI*'s notation aRb, we introduced the notation `R a b`, which we use when `R` has type `Rel A B`, `a` has type `A`, and `b` has type `B`. (The notation `(a, b) ∈ R`, in contrast, can be used only when `R` has type `Set (A × B)`.) If `R` has type `Rel A B`, then we think of `R` as representing some relationship that might hold between `a` and `b`, and `R a b` as the proposition saying that this relationship holds. And although `R` is not a set of ordered pairs, there is a corresponding set, `extension R`, of type `Set (A × B)`, with the property that `(a, b) ∈ extension R` if and only if `R a b`.

We will take a similar approach to functions in this chapter. For any types `A` and `B`, we introduce a new type `A → B`. If `f` has type `A → B`, then we think of `f` as representing some operation that can be applied to an object of type `A` to produce a corresponding object of type `B`. We will say that `f` is a *function* from `A` to `B`, and `A` is the *domain* of `f`. If `a` has type `A`, then we write `f a` (with a space but no parentheses) for the result of applying the operation represented by `f` to the object `a`. Thus, if we have `f : A → B` and `a : A`, then `f a` has type `B`. As with relations, if `f` has type `A → B`, then `f` is not a set of ordered pairs. But there is a corresponding set of ordered pairs, which we will call the *graph* of `f`, whose elements are the ordered pairs `(a, b)` for which `f a = b`:

```
def graph {A B : Type} (f : A → B) : Set (A × B) :=
    { (a, b) : A × B | f a = b }
```

```
theorem simp_graph {A B : Type} (f : A → B) (a : A) (b : B) :
    (a, b) ∈ graph f ↔ f a = b := by rfl
```

Every set of type `Set (A × B)` is the extension of some relation from `A` to `B`, but not every such set is the graph of a function from `A` to `B`. To be the graph of a function, it must have the property that was used to define functions in *HTPI*: each object of type `A` must be paired in the set with exactly one object of type `B`. Let's give this property a name:

```
def is_func_graph {A B : Type} (F : Set (A × B)) : Prop :=
    ∀ (x : A), ∃! (y : B), (x, y) ∈ F
```

And now we can say that the sets of type `Set (A × B)` that are graphs of functions from `A` to `B` are precisely the ones that have the property `is_func_graph`:

```
theorem func_from_graph {A B : Type} (F : Set (A × B)) :
    (∃ (f : A → B), graph f = F) ↔ is_func_graph F
```

We will ask you to prove the left-to-right direction of this theorem in the exercises. The right-to-left direction can also be proven in Lean, but the proof requires methods that go beyond the scope of this book. Nevertheless, we will go ahead and use the theorem when we find it useful.

Section 5.1 of *HTPI* proves two theorems about functions. The first gives a convenient way of proving that two functions are equal:

Theorem 5.1.4. *Suppose f and g are functions from A to B. If $\forall a \in A(f(a) = g(a))$, then $f = g$.*

The proof of this theorem in *HTPI* is based on the axiom of extensionality for sets. But in Lean, functions aren't sets of ordered pairs, so this method of proof won't work. Fortunately, Lean has a similar axiom of extensionality for functions. The axiom is called `funext`, and it proves Theorem 5.1.4.

```
theorem Theorem_5_1_4 {A B : Type} (f g : A → B) :
    (∀ (a : A), f a = g a) → f = g := funext
```

We saw previously that if we are trying to prove `X = Y`, where `X` and `Y` both have type `Set U`, then often the best first step is the tactic `apply Set.ext`, which converts the goal to `∀ (x : U), x ∈ X ↔ x ∈ Y`. Similarly, if we are trying to prove `f = g`, where `f` and `g` both have type `A → B`, then we will usually start with the tactic `apply funext`, which will convert the goal to `∀ (x : A), f x = g x`. By Theorem_5_1_4, this implies the original goal `f = g`. For example, here is a proof that if two functions have the same graph, then they are equal:

```
example {A B : Type} (f g : A → B) :
    graph f = graph g → f = g := by
  assume h1 : graph f = graph g  --Goal: f = g
  apply funext                    --Goal: ∀ (x : A), f x = g x
  fix x : A
  have h2 : (x, f x) ∈ graph f := by
    define                        --Goal: f x = f x
    rfl
    done
  rewrite [h1] at h2              --h2: (x, f x) ∈ graph g
  define at h2                    --h2: g x = f x
  show f x = g x from h2.symm
  done
```

The axiom of extensionality for sets says that a set is completely determined by its elements. This is what justifies our usual method of defining a set: we specify what its elements are, using notation like {0, 1, 2} or { x : Nat | x < 3 }. Similarly, the axiom of extensionality for functions says that a function is completely determined by its values, and therefore we can define a function by specifying its values. For instance, we can define a function from Nat to Nat by specifying, for any n : Nat, the result of applying the function to n. As an example of this, we could define the "squaring function" from Nat to Nat to be the function that, when applied to any n : Nat, produces the result n ^ 2. Here are two ways to define this function in Lean:

```
def square1 (n : Nat) : Nat := n ^ 2


def square2 : Nat → Nat := fun (n : Nat) => n ^ 2
```

The first of these definitions uses notation we have used before; it says that if n has type Nat, then the expression square1 n also has type Nat, and it is definitionally equal to n ^ 2. The second definition introduces new Lean notation. It says that square2 has type Nat → Nat, and it defines it to be the function that, when applied to any n of type Nat, yields the result n ^ 2. Of course, this also means that square2 n is definitionally equal to n ^ 2. In general, the notation fun (x : A) => ... means "the function which, when applied to any x of type A, yields the result …" The two definitions above are equivalent. You can ask Lean to confirm this, and try out the squaring function, as follows (the #eval command asks Lean to evaluate an expression):

```
example : square1 = square2 := by rfl


#eval square1 7      --Answer: 49
```

There is one more theorem in Section 5.1 of *HTPI*. Theorem 5.1.5 says that if f is a function

from A to B and g is a function from B to C, then the composition of g and f is a function from A to C. To state this theorem in Lean, we will have to make adjustments for the differences between the treatment of functions in *HTPI* and Lean. In Chapter 4, we defined comp S R to be the composition of S and R, where R has type Set (A × B) and S had type Set (B × C). But functions in Lean are not sets of ordered pairs, so we cannot apply the operation comp to them. We can, however, apply it to their graphs. So the theorem corresponding to Theorem 5.1.5 in *HTPI* is this:

```
theorem Theorem_5_1_5 {A B C : Type} (f : A → B) (g : B → C) :
    ∃ (h : A → C), graph h = comp (graph g) (graph f) := by
  let h : A → C := fun (x : A) => g (f x)
  apply Exists.intro h
  apply Set.ext
  fix (a, c) : A × C
  apply Iff.intro
  · -- Proof that (a, c) ∈ graph h → (a, c) ∈ comp (graph g) (graph f)
    assume h1 : (a, c) ∈ graph h
    define at h1  --h1: h a = c
    define        --Goal: ∃ (x : B), (a, x) ∈ graph f ∧ (x, c) ∈ graph g
    apply Exists.intro (f a)
    apply And.intro
    · -- Proof that (a, f a) ∈ graph f
      define
      rfl
      done
    · -- Proof that (f a, c) ∈ graph g
      define
      show g (f a) = c from h1
      done
    done
  · -- Proof that (a, c) ∈ comp (graph g) (graph f) → (a, c) ∈ graph h
    assume h1 : (a, c) ∈ comp (graph g) (graph f)
    define        --Goal: h a = c
    define at h1  --h1: ∃ (x : B), (a, x) ∈ graph f ∧ (x, c) ∈ graph g
    obtain (b : B) (h2 : (a, b) ∈ graph f ∧ (b, c) ∈ graph g) from h1
    have h3 : (a, b) ∈ graph f := h2.left
    have h4 : (b, c) ∈ graph g := h2.right
    define at h3          --h3: f a = b
    define at h4          --h4: g b = c
    rewrite [←h3] at h4   --h4: g (f a) = c
    show h a = c from h4
    done
```

```
    done
```

Notice that the proof of `Theorem_5_1_5` begins by defining the function `h` for which `graph h = comp (graph g) (graph f)`. The definition says that for all `x` of type `A`, `h x = g (f x)`. This function `h` is called the *composition* of `g` and `f`, and it is denoted `g ∘ f`. (To type `∘` in VSCode, type `\comp` or `\circ`.) In other words, `g ∘ f` has type `A → C`, and for all `x` of type `A`, `(g ∘ f) x` is definitionally equal to `g (f x)`. In *HTPI*, functions are sets of ordered pairs, and the operation of composition of functions is literally the same as the operation `comp` that we used in Chapter 4. But in Lean, we distinguish among functions, relations, and sets of ordered pairs, so all we can say is that the operation of composition of functions corresponds to the operation `comp` from Chapter 4. The correspondence is that, as shown in the proof of `Theorem_5_1_5`, if `h = g ∘ f`, then `graph h = comp (graph g) (graph f)`.

We saw in part 4 of Theorem 4.2.5 that composition of relations is associative. Composition of functions is also associative. In fact, if `f : A → B`, `g : B → C`, and `h : C → D`, then `h ∘ (g ∘ f)` and `(h ∘ g) ∘ f` are definitionally equal, since they both mean the same thing as `fun (x : A) => h (g (f a))`. As a result, the tactic `rfl` proves the associativity of composition of functions:

```
example {A B C D : Type} (f : A → B) (g : B → C) (h : C → D) :
    h ∘ (g ∘ f) = (h ∘ g) ∘ f := by rfl
```

HTPI defines the identity function on a set A to be the function i_A from A to A such that $\forall x \in A(i_A(x) = x)$, and Exercise 9 from Section 4.3 of *HTPI* implies that if $f : A \to B$, then $f \circ i_A = f$ and $i_B \circ f = f$. We say, therefore, that the identity functions are *identity elements* for composition of functions. Similarly, in Lean, for each type `A` there is an identity function from `A` to `A`. This identity function is denoted `id`; there is no need to specify `A` in the notation, because `A` is an implicit argument to `id`. Thus, when you use `id` to denote an identity function, Lean will figure out what type `A` to use as the domain of the function. (If, for some reason, you want to specify that the domain is some type `A`, you can write `@id A` instead of `id`.) For any `x`, of any type, `id x` is definitionally equal to `x`, and as a result the proof that `id` is an identity element for composition of functions can also be done with the `rfl` tactic:

```
example {A B : Type} (f : A → B) : f ∘ id = f := by rfl
```

```
example {A B : Type} (f : A → B) : id ∘ f = f := by rfl
```

Exercises

1.
```
theorem func_from_graph_ltr {A B : Type} (F : Set (A × B)) :
    (∃ (f : A → B), graph f = F) → is_func_graph F := sorry
```

2.
```
theorem Exercise_5_1_13a
    {A B C : Type} (R : Set (A × B)) (S : Set (B × C)) (f : A → C)
    (h1 : ∀ (b : B), b ∈ Ran R ∧ b ∈ Dom S) (h2 : graph f = comp S R) :
    is_func_graph S := sorry
```

3.
```
theorem Exercise_5_1_14a
    {A B : Type} (f : A → B) (R : BinRel A) (S : BinRel B)
    (h : ∀ (x y : A), R x y ↔ S (f x) (f y)) :
    reflexive S → reflexive R := sorry
```

4. Here is a putative theorem:

Theorem?. *Suppose $f : A \to B$, R is a binary relation on A, and S is the binary relation on B defined as follows:*

$$\forall x \in B \forall y \in B(xSy \leftrightarrow \exists u \in A \exists v \in A(f(u) = x \land f(v) = y \land uRv)).$$

If R is reflexive then S is reflexive.

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

```
--You might not be able to complete this proof
theorem Exercise_5_1_15a
    {A B : Type} (f : A → B) (R : BinRel A) (S : BinRel B)
    (h : ∀ (x y : B), S x y ↔ ∃ (u v : A), f u = x ∧ f v = y ∧ R u v) :
    reflexive R → reflexive S := sorry
```

5. Here is a putative theorem with an incorrect proof:

Theorem?. *Suppose $f : A \to B$, R is a binary relation on A, and S is the binary relation on B defined as follows:*

$$\forall x \in B \forall y \in B(xSy \leftrightarrow \exists u \in A \exists v \in A(f(u) = x \land f(v) = y \land uRv)).$$

If R is transitive then S is transitive.

Incorrect Proof. Suppose R is transitive. Let x, y, and z be arbitrary elements of B. Assume that xSy and ySz. By the definition of S, this means that there are u, v, and w in A such that $f(u) = x$, $f(v) = y$, $f(w) = z$, uRv, and vRw. Since R is transitive, it follows that uRw. Since $f(u) = x$, $f(w) = z$, and uRw, xSz. Therefore S is transitive. □

Find the mistake in the proof by attempting to write the proof in Lean. Is the theorem correct?

```
--You might not be able to complete this proof
theorem Exercise_5_1_15c
    {A B : Type} (f : A → B) (R : BinRel A) (S : BinRel B)
    (h : ∀ (x y : B), S x y ↔ ∃ (u v : A), f u = x ∧ f v = y ∧ R u v) :
    transitive R → transitive S := sorry
```

6.
```
theorem Exercise_5_1_16b
    {A B : Type} (R : BinRel B) (S : BinRel (A → B))
    (h : ∀ (f g : A → B), S f g ↔ ∀ (x : A), R (f x) (g x)) :
    symmetric R → symmetric S := sorry
```

7.
```
theorem Exercise_5_1_17a {A : Type} (f : A → A) (a : A)
    (h : ∀ (x : A), f x = a) : ∀ (g : A → A), f ∘ g = f := sorry
```

8.
```
theorem Exercise_5_1_17b {A : Type} (f : A → A) (a : A)
    (h : ∀ (g : A → A), f ∘ g = f) :
    ∃ (y : A), ∀ (x : A), f x = y := sorry
```

5.2. One-to-One and Onto

Section 5.2 of *HTPI* introduces two important properties that a function might have. A function `f : A → B` is called *onto* if for every `b` of type `B` there is at least one `a` of type `A` such that `f a = b`:

```
def onto {A B : Type} (f : A → B) : Prop :=
    ∀ (y : B), ∃ (x : A), f x = y
```

It is called *one-to-one* if there do *not* exist distinct `a1` and `a2` of type `A` such that `f a1 = f a2`. This phrasing of the definition makes it clear what is at issue: Are there distinct objects in the domain to which the function assigns the same value? But it is a negative statement, and that would make it difficult to work with it in proofs. Fortunately, it is not hard to rephrase

the definition as an equivalent positive statement, using quantifier negation, De Morgan, and conditional laws. The resulting equivalent positive statement is given in Theorem 5.2.3 of *HTPI*, and we take it as our official definition of `one_to_one` in Lean:

```lean
def one_to_one {A B : Type} (f : A → B) : Prop :=
    ∀ (x1 x2 : A), f x1 = f x2 → x1 = x2
```

There is only one more theorem about these properties in Section 5.2 of *HTPI*. It says that a composition of one-to-one functions is one-to-one, and a composition of onto functions is onto. It is straightforward to carry out these proofs in Lean by simplying applying the definitions of the relevant concepts.

```lean
theorem Theorem_5_2_5_1 {A B C : Type} (f : A → B) (g : B → C) :
    one_to_one f → one_to_one g → one_to_one (g ∘ f) := by
  assume h1 : one_to_one f
  assume h2 : one_to_one g
  define at h1  --h1: ∀ (x1 x2 : A), f x1 = f x2 → x1 = x2
  define at h2  --h2: ∀ (x1 x2 : B), g x1 = g x2 → x1 = x2
  define        --Goal: ∀ (x1 x2 : A), (g ∘ f) x1 = (g ∘ f) x2 → x1 = x2
  fix a1 : A
  fix a2 : A    --Goal: (g ∘ f) a1 = (g ∘ f) a2 → a1 = a2
  define : (g ∘ f) a1; define : (g ∘ f) a2
                --Goal: g (f a1) = g (f a2) → a1 = a2
  assume h3 : g (f a1) = g (f a2)
  have h4 : f a1 = f a2 := h2 (f a1) (f a2) h3
  show a1 = a2 from h1 a1 a2 h4
  done
```

Notice that the tactic `define : (g ∘ f) a1` replaces `(g ∘ f) a1` with its definition, `g (f a1)`. As usual, this step isn't really needed—Lean will apply the definition on its own when necessary, without being told. But using this tactic makes the proof easier to read. We use a similar approach for the second part of the theorem.

```lean
theorem Theorem_5_2_5_2 {A B C : Type} (f : A → B) (g : B → C) :
    onto f → onto g → onto (g ∘ f) := by
  assume h1 : onto f
  assume h2 : onto g
  define at h1         --h1: ∀ (y : B), ∃ (x : A), f x = y
  define at h2         --h2: ∀ (y : C), ∃ (x : B), g x = y
  define               --Goal: ∀ (y : C), ∃ (x : A), (g ∘ f) x = y
  fix c : C
  obtain (b : B) (h3 : g b = c) from h2 c
```

```
    obtain (a : A) (h4 : f a = b) from h1 b
    apply Exists.intro a    --Goal: (g ∘ f) a = c
    define : (g ∘ f) a      --Goal: g (f a) = c
    rewrite [←h4] at h3
    show g (f a) = c from h3
    done
```

Exercises

1.
```
theorem Exercise_5_2_10a {A B C : Type} (f: A → B) (g : B → C) :
    onto (g ∘ f) → onto g := sorry
```

2.
```
theorem Exercise_5_2_10b {A B C : Type} (f: A → B) (g : B → C) :
    one_to_one (g ∘ f) → one_to_one f := sorry
```

3.
```
theorem Exercise_5_2_11a {A B C : Type} (f: A → B) (g : B → C) :
    onto f → ¬(one_to_one g) → ¬(one_to_one (g ∘ f)) := sorry
```

4.
```
theorem Exercise_5_2_11b {A B C : Type} (f: A → B) (g : B → C) :
    ¬(onto f) → one_to_one g → ¬(onto (g ∘ f)) := sorry
```

5.
```
theorem Exercise_5_2_12 {A B : Type} (f : A → B) (g : B → Set A)
    (h : ∀ (b : B), g b = { a : A | f a = b }) :
    onto f → one_to_one g := sorry
```

6.
```
theorem Exercise_5_2_16 {A B C : Type}
    (R : Set (A × B)) (S : Set (B × C)) (f : A → C) (g : B → C)
    (h1 : graph f = comp S R) (h2 : graph g = S) (h3 : one_to_one g) :
    is_func_graph R := sorry
```

7.
```
theorem Exercise_5_2_17a
    {A B : Type} (f : A → B) (R : BinRel A) (S : BinRel B)
    (h1 : ∀ (x y : B), S x y ↔ ∃ (u v : A), f u = x ∧ f v = y ∧ R u v)
    (h2 : onto f) : reflexive R → reflexive S := sorry
```

8.   ```
     theorem Exercise_5_2_17b
         {A B : Type} (f : A → B) (R : BinRel A) (S : BinRel B)
         (h1 : ∀ (x y : B), S x y ↔ ∃ (u v : A), f u = x ∧ f v = y ∧ R u v)
         (h2 : one_to_one f) : transitive R → transitive S := sorry
     ```

9.   ```
     theorem Exercise_5_2_21a {A B C : Type} (f : B → C) (g h : A → B)
         (h1 : one_to_one f) (h2 : f ∘ g = f ∘ h) : g = h := sorry
     ```

10.  ```
     theorem Exercise_5_2_21b {A B C : Type} (f : B → C) (a : A)
         (h1 : ∀ (g h : A → B), f ∘ g = f ∘ h → g = h) :
         one_to_one f := sorry
     ```

5.3. Inverses of Functions

Section 5.3 of *HTPI* is motivated by the following question: If f is a function from A to B, is f^{-1} a function from B to A? Here is the first theorem in that section:

Theorem 5.3.1. *Suppose $f : A \to B$. If f is one-to-one and onto, then $f^{-1} : B \to A$.*

Of course, we will have to rephrase this theorem slightly to prove it in Lean. If `f` has type `A → B`, then the inverse operation `inv` cannot be applied to `f`, but it can be applied to `graph f`. So we must rephrase the theorem like this:

```
theorem Theorem_5_3_1 {A B : Type}
    (f : A → B) (h1 : one_to_one f) (h2 : onto f) :
    ∃ (g : B → A), graph g = inv (graph f)
```

To prove this theorem, we will use the theorem `func_from_graph` that was stated in Section 5.1. We can remind ourselves of what that theorem says by using the command `#check @func_from_graph`, which gives the result:

```
@func_from_graph : ∀ {A B : Type} (F : Set (A × B)),
                   (∃ (f : A → B), graph f = F) ↔ is_func_graph F
```

This means that, in the context of the proof of `Theorem_5_3_1`, `func_from_graph (inv (graph f))` is a proof of the statement `∃ (g : B → A), graph g = inv (graph f) ↔ is_func_graph (inv (graph f))`. Therefore the tactic `rewrite [func_from_graph (inv (graph f))]` will change the goal to `is_func_graph (inv (func f))`. In fact, we can just use `rewrite [func_from_graph]`,

and Lean will figure out how to apply the theorem to rewrite the goal. The rest of the proof is straightforward.

```
theorem Theorem_5_3_1 {A B : Type}
    (f : A → B) (h1 : one_to_one f) (h2 : onto f) :
    ∃ (g : B → A), graph g = inv (graph f) := by
  rewrite [func_from_graph]   --Goal: is_func_graph (inv (graph f))
  define        --Goal: ∀ (x : B), ∃! (y : A), (x, y) ∈ inv (graph f)
  fix b : B
  exists_unique
  · -- Existence
    define at h2           --h2: ∀ (y : B), ∃ (x : A), f x = y
    obtain (a : A) (h4 : f a = b) from h2 b
    apply Exists.intro a  --Goal: (b, a) ∈ inv (graph f)
    define                 --Goal: f a = b
    show f a = b from h4
    done
  · -- Uniqueness
    fix a1 : A; fix a2 : A
    assume h3 : (b, a1) ∈ inv (graph f)
    assume h4 : (b, a2) ∈ inv (graph f) --Goal: a1 = a2
    define at h3           --h3: f a1 = b
    define at h4           --h4: f a2 = b
    rewrite [←h4] at h3   --h3: f a1 = f a2
    define at h1           --h1: ∀ (x1 x2 : A), f x1 = f x2 → x1 = x2
    show a1 = a2 from h1 a1 a2 h3
    done
  done
```

Suppose, now, that we have f : A → B, g : B → A, and graph g = inv (graph f), as in Theorem_5_3_1. What can we say about the relationship between f and g? One answer is that g ∘ f = id and f ∘ g = id, as shown in Theorem 5.3.2 of *HTPI*. We'll prove one of these facts, and leave the other as an exercise for you.

```
theorem Theorem_5_3_2_1 {A B : Type} (f : A → B) (g : B → A)
    (h1 : graph g = inv (graph f)) : g ∘ f = id := by
  apply funext            --Goal: ∀ (x : A), (g ∘ f) x = id x
  fix a : A               --Goal: (g ∘ f) a = id a
  have h2 : (f a, a) ∈ graph g := by
    rewrite [h1]          --Goal: (f a, a) ∈ inv (graph f)
    define                --Goal: f a = f a
    rfl
```

```
    done
  define at h2              --h2: g (f a) = a
  show (g ∘ f) a = id a from h2
  done

theorem Theorem_5_3_2_2 {A B : Type} (f : A → B) (g : B → A)
    (h1 : graph g = inv (graph f)) : f ∘ g = id := sorry
```

Combining the theorems above, we have shown that if f is one-to-one and onto, then there is a function g such that g ∘ f = id and f ∘ g = id. In fact, the converse is true as well: if such a function g exists, then f must be one-to-one and onto. Again, we'll prove one statement as leave the second as an exercise.

```
theorem Theorem_5_3_3_1 {A B : Type} (f : A → B) :
    (∃ (g : B → A), g ∘ f = id) → one_to_one f := by
  assume h1 : ∃ (g : B → A), g ∘ f = id
  obtain (g : B → A) (h2 : g ∘ f = id) from h1
  define               --Goal: ∀ (x1 x2 : A), f x1 = f x2 → x1 = x2
  fix a1 : A; fix a2 : A
  assume h3 : f a1 = f a2
  show a1 = a2 from
    calc a1
        = id a1 := by rfl
      _ = (g ∘ f) a1 := by rw [h2]
      _ = g (f a1) := by rfl
      _ = g (f a2) := by rw [h3]
      _ = (g ∘ f) a2 := by rfl
      _ = id a2 := by rw [h2]
      _ = a2 := by rfl
  done

theorem Theorem_5_3_3_2 {A B : Type} (f : A → B) :
    (∃ (g : B → A), f ∘ g = id) → onto f := sorry
```

We can combine the theorems above to show that if we have f : A → B, g : B → A, g ∘ f = id, and f ∘ g = id, then graph g must be the inverse of graph f. Compare the proof below to the proof of Theorem 5.3.5 in *HTPI*.

```
theorem Theorem_5_3_5 {A B : Type} (f : A → B) (g : B → A)
    (h1 : g ∘ f = id) (h2 : f ∘ g = id) : graph g = inv (graph f) := by
  have h3 : one_to_one f := Theorem_5_3_3_1 f (Exists.intro g h1)
  have h4 : onto f := Theorem_5_3_3_2 f (Exists.intro g h2)
```

```
    obtain (g' : B → A) (h5 : graph g' = inv (graph f))
      from Theorem_5_3_1 f h3 h4
    have h6 : g' ∘ f = id := Theorem_5_3_2_1 f g' h5
    have h7 : g = g' :=
      calc g
          = id ∘ g := by rfl
        _ = (g' ∘ f) ∘ g := by rw [h6]
        _ = g' ∘ (f ∘ g) := by rfl
        _ = g' ∘ id := by rw [h2]
        _ = g' := by rfl
    rewrite [←h7] at h5
    show graph g = inv (graph f) from h5
    done
```

Exercises

1.
```
    theorem Theorem_5_3_2_2 {A B : Type} (f : A → B) (g : B → A)
        (h1 : graph g = inv (graph f)) : f ∘ g = id := sorry
```

2.
```
    theorem Theorem_5_3_3_2 {A B : Type} (f : A → B) :
        (∃ (g : B → A), f ∘ g = id) → onto f := sorry
```

3.
```
    theorem Exercise_5_3_11a {A B : Type} (f : A → B) (g : B → A) :
        one_to_one f → f ∘ g = id → graph g = inv (graph f) := sorry
```

4.
```
    theorem Exercise_5_3_11b {A B : Type} (f : A → B) (g : B → A) :
        onto f → g ∘ f = id → graph g = inv (graph f) := sorry
```

5.
```
    theorem Exercise_5_3_14a {A B : Type} (f : A → B) (g : B → A)
        (h : f ∘ g = id) : ∀ x ∈ Ran (graph g), g (f x) = x := sorry
```

6.
```
    theorem Exercise_5_3_18 {A B C : Type} (f : A → C) (g : B → C)
        (h1 : one_to_one g) (h2 : onto g) :
        ∃ (h : A → B), g ∘ h = f := sorry
```

The next two exercises will use the following definition:

```
   def conj (A : Type) (f1 f2 : A → A) : Prop :=
       ∃ (g g' : A → A), (f1 = g' ∘ f2 ∘ g) ∧ (g ∘ g' = id) ∧ (g' ∘ g = id)
```

7.   ```theorem Exercise_5_3_17a {A : Type} : symmetric (conj A) := sorry```

8.
```
       theorem Exercise_5_3_17b {A : Type} (f1 f2 : A → A)
           (h1 : conj A f1 f2) (h2 : ∃ (a : A), f1 a = a) :
           ∃ (a : A), f2 a = a := sorry
```

5.4. Closures

Suppose we have `f : A → A` and `C : Set A`. We say that `C` is *closed* under `f` if the value of `f` at any element of `C` is again an element of `C`:

```
   def closed {A : Type} (f : A → A) (C : Set A) : Prop := ∀ x ∈ C, f x ∈ C
```

According to this definition, `closed f C` means that `C` is closed under `f`. Sometimes, if we have a set `B` of type `Set A` that is not closed under `f`, we are interested in adding more elements to the set to make it closed. The *closure* of `B` under `f` is the smallest set containing `B` that is closed under `f`. That is, it is the smallest element of `{ D : Set A | B ⊆ D ∧ closed f D }`, where we use the subset partial ordering on `Set A` to determine which element is smallest. We will write `closure f B C` to mean that the closure of `B` under `f` is `C`. We can define this as follows:

```
   def closure {A : Type} (f : A → A) (B C : Set A) : Prop :=
       smallestElt (sub A) C { D : Set A | B ⊆ D ∧ closed f D }
```

We know that smallest elements, when they exist, are unique, so it makes sense to talk about *the* closure of `B` under `f`. But not every set has a smallest element. Does every set have a closure? Theorem 5.4.5 in *HTPI* says that the answer is yes. The idea behind the proof is that, for any family of sets `F`, if `F` has a smallest element under the subset partial order, then that smallest element is equal to $\cap_0 F$. (We'll ask you to prove this in the exercises.)

```
   theorem Theorem_5_4_5 {A : Type} (f : A → A) (B : Set A) :
       ∃ (C : Set A), closure f B C := by
     let F : Set (Set A) := { D : Set A | B ⊆ D ∧ closed f D }
     let C : Set A := ∩₀F
     apply Exists.intro C    --Goal: closure f B C
```

```
define                   --Goal: C ∈ F ∧ ∀ x ∈ F, C ⊆ x
apply And.intro
· -- Proof that C ∈ F
  define                  --Goal: B ⊆ C ∧ closed f C
  apply And.intro
  · -- Proof that B ⊆ C
    fix a : A
    assume h1 : a ∈ B       --Goal: a ∈ C
    define                  --Goal: ∀ t ∈ F, a ∈ t
    fix D : Set A
    assume h2 : D ∈ F
    define at h2            --h2: B ⊆ D ∧ closed f D
    show a ∈ D from h2.left h1
    done
  · -- Proof that C is closed under f
    define                  --Goal: ∀ x ∈ C, f x ∈ C
    fix a : A
    assume h1 : a ∈ C       --Goal: f a ∈ C
    define                  --Goal: ∀ t ∈ F, f a ∈ t
    fix D : Set A
    assume h2 : D ∈ F       --Goal: f a ∈ D
    define at h1            --h1: ∀ t ∈ F, a ∈ t
    have h3 : a ∈ D := h1 D h2
    define at h2            --h2: B ⊆ D ∧ closed f D
    have h4 : closed f D := h2.right
    define at h4            --h4: ∀ x ∈ D, f x ∈ D
    show f a ∈ D from h4 a h3
    done
  done
· -- Proof that C is smallest
  fix D : Set A
  assume h1 : D ∈ F       --Goal: sub A C D
  define
  fix a : A
  assume h2 : a ∈ C       --Goal: a ∈ D
  define at h2            --h2: ∀ t ∈ F, a ∈ t
  show a ∈ D from h2 D h1
  done
done
```

The idea of the closure of a set under a function can also be applied to functions of two variables. One way to represent a function of two variables on a type A would be to use a

function g of type (A × A) → A. If a and b have type A, then (a, b) has type A × A, and the result of applying the function g to the pair of values a and b would be written g (a, b).

However, there is another way to represent a function of two variables that turns out to be more convenient in Lean. Suppose f has type A → A → A. As with the arrow used in conditional propositions, the arrow for function types groups to the right, so A → A → A means A → (A → A). Thus, if a has type A, then f a has type A → A. In other words, f a is a function from A to A, and therefore if b has type A then f a b has type A. The upshot is that if f is followed by two objects of type A, then the resulting expression has type A, so f can be thought of as a function that applies to a pair of objects of type A and gives a value of type A.

For example, we can think of addition of integers as a function of two variables. Here are three ways to define this function in Lean.

```
def plus (m n : Int) : Int := m + n

def plus' : Int → Int → Int := fun (m n : Int) => m + n

def plus'' : Int → Int → Int := fun (m : Int) => (fun (n : Int) => m + n)
```

The third definition matches the description above most closely: plus'' is a function that, when applied to an integer m, produces a new function plus'' m : Int → Int. The function plus'' m is defined to be the function that, when applied to an integer n, produces the value m + n. In other words, plus'' m n = m + n. The first two definitions are more convenient ways of defining exactly the same function. Let's have Lean confirm this, and try out the function:

```
example : plus = plus'' := by rfl

example : plus' = plus'' := by rfl

#eval plus 3 2      --Answer: 5
```

There are two reasons why this way of representing functions of two variables in Lean is more convenient. First, it saves us the trouble of grouping the arguments of the function together into an ordered pair before applying the function. If we have f : A → A → A and a b : A, then to apply the function f to the arguments a and b we can just write f a b. Second, it allows for the possibility of "partially applying" the function f. The expression f a is meaningful, and denotes the function that, when applied to any b : A, produces the result f a b. For example, if m is an integer, then plus m denotes the function that, when applied to an integer n, produces the result m + n. We might call plus m the "add to m" function.

We have actually been using these ideas for a long time. In Chapter 3, we introduced the type Pred U of predicates applying to objects of type U, but we did not explain how such predicates are represented internally in Lean. In fact, Pred U is defined to be the type U → Prop, so if P

has type `Pred U`, then `P` is a function from `U` to `Prop`, and if `x` has type `U`, then the proposition `P x` is the result of applying the function `P` to `x`. Similarly, `Rel A B` stands for `A → B → Prop`, so if `R` has type `Rel A B`, then `R` is a function of two variables, one of type `A` and one of type `B`. Earlier in this section, we defined `closed f C` to be the proposition asserting that `C` is closed under `f`. This means that `closed` is a function of two variables, the first a function `f` of type `A → A` and the second a set `C` of type `Set A` (where the type `A` is an implicit argument of `closed`). But that means that the partial application `closed f` denotes a function from `Set A` to `Prop`. In other words, `closed f` is a predicate applying to sets of type `Set A`; we could think of it as the "is closed under `f`" predicate. Similarly, in Section 4.4 we defined `sub` to be a function of three variables: if `A` is a type and `X` and `Y` have type `Set A`, then `sub A X Y` is the proposition `X ⊆ Y`. Since then, we have used the partial application `sub A`, which is the subset relation on `Set A`. For example, we used it earlier in this section in the definition of `closure`.

Returning to the subject of closures, here's how we can extend the idea of closures to functions of two variables:

```
def closed2 {A : Type} (f : A → A → A) (C : Set A) : Prop :=
    ∀ x ∈ C, ∀ y ∈ C, f x y ∈ C

def closure2 {A : Type} (f : A → A → A) (B C : Set A) : Prop :=
    smallestElt (sub A) C { D : Set A | B ⊆ D ∧ closed2 f D }
```

We will leave it as an exercise for you to prove that closures under functions of two variables also exist.

```
theorem Theorem_5_4_9 {A : Type} (f : A → A → A) (B : Set A) :
    ∃ (C : Set A), closure2 f B C := sorry
```

Exercises

1.
```
example {A : Type} (F : Set (Set A)) (B : Set A) :
    smallestElt (sub A) B F → B = ⋂₀F := sorry
```

2. If `B` has type `Set A`, then `complement B` is the set `{ a : A | a ∉ B }`. Thus, for any `a` of type `A`, `a ∈ complement B` if and only if `a ∉ B`:

```
def complement {A : Type} (B : Set A) : Set A := { a : A | a ∉ B }

theorem simp_complement {A : Type} (a : A) (B : Set A) :
    a ∈ complement B ↔ a ∉ B := by rfl
```

Use this definition and theorem to prove the following theorem:

```
theorem Exercise_5_4_7 {A : Type} (f g : A → A) (C : Set A)
    (h1 : f ∘ g = id) (h2 : closed f C) : closed g (complement C) := sorry
```

3.
```
theorem Exercise_5_4_9a {A : Type} (f : A → A) (C1 C2 : Set A)
    (h1 : closed f C1) (h2 : closed f C2) : closed f (C1 ∪ C2) := sorry
```

4.
```
theorem Exercise_5_4_10a {A : Type} (f : A → A) (B1 B2 C1 C2 : Set A)
    (h1 : closure f B1 C1) (h2 : closure f B2 C2) :
    B1 ⊆ B2 → C1 ⊆ C2 := sorry
```

5.
```
theorem Exercise_5_4_10b {A : Type} (f : A → A) (B1 B2 C1 C2 : Set A)
    (h1 : closure f B1 C1) (h2 : closure f B2 C2) :
    closure f (B1 ∪ B2) (C1 ∪ C2) := sorry
```

6.
```
theorem Theorem_5_4_9 {A : Type} (f : A → A → A) (B : Set A) :
    ∃ (C : Set A), closure2 f B C := sorry
```

7. Suppose we define a set to be closed under a family of functions if it is closed under all of the functions in the family. Of course, the closure of a set `B` under a family of functions is the smallest set containing `B` that is closed under the family.

```
def closed_family {A : Type} (F : Set (A → A)) (C : Set A) : Prop :=
    ∀ f ∈ F, closed f C

def closure_family {A : Type} (F : Set (A → A)) (B C : Set A) : Prop :=
    smallestElt (sub A) C { D : Set A | B ⊆ D ∧ closed_family F D }
```

Prove that the closure of a set under a family of functions always exists:

```
theorem Exercise_5_4_13a {A : Type} (F : Set (A → A)) (B : Set A) :
    ∃ (C : Set A), closure_family F B C := sorry
```

5.5. Images and Inverse Images: A Research Project

Section 5.5 of *HTPI* introduces two new definitions. Suppose $f : A \to B$. If $X \subseteq A$, then the *image* of X under f is the set $f(X)$ defined as follows:

$$f(X) = \{f(x) \mid x \in X\} = \{b \in B \mid \exists x \in X (f(x) = b)\}.$$

If $Y \subseteq B$, then the *inverse image* of Y under f is the set $f^{-1}(Y)$ defined as follows:

$$f^{-1}(Y) = \{a \in A \mid f(a) \in Y\}.$$

Here are definitions of these concepts in Lean:

```
def image {A B : Type} (f : A → B) (X : Set A) : Set B :=
    { f x | x ∈ X }

def inverse_image {A B : Type} (f : A → B) (Y : Set B) : Set A :=
    { a : A | f a ∈ Y }

-- The following theorems illustrate the meaning of these definitions:
theorem simp_image {A B : Type} (f : A → B) (X : Set A) (b : B) :
    b ∈ image f X ↔ ∃ x ∈ X, f x = b := by rfl

theorem simp_inverse_image {A B : Type} (f : A → B) (Y : Set B) (a : A) :
    a ∈ inverse_image f Y ↔ f a ∈ Y := by rfl
```

It is natural to wonder how these concepts interact with familiar operations on sets. *HTPI* gives an example of such an interaction in Theorem 5.5.2. The theorem makes two assertions. Here are proofs of the two parts of the theorem in Lean.

```
theorem Theorem_5_5_2_1 {A B : Type} (f : A → B) (W X : Set A) :
    image f (W ∩ X) ⊆ image f W ∩ image f X := by
  fix y : B
  assume h1 : y ∈ image f (W ∩ X)  --Goal: y ∈ image f W ∩ image f X
  define at h1                      --h1: ∃ (x : A), x ∈ W ∩ X ∧ f x = y
  obtain (x : A) (h2 : x ∈ W ∩ X ∧ f x = y) from h1
  define : x ∈ W ∩ X at h2          --h2: (x ∈ W ∧ x ∈ X) ∧ f x = y
  apply And.intro
  · -- Proof that y ∈ image f W
    define                          --Goal: ∃ (x : A), x ∈ W ∧ f x = y
    show ∃ (x : A), x ∈ W ∧ f x = y from
      Exists.intro x (And.intro h2.left.left h2.right)
    done
```

```
    · -- Proof that y ∈ image f X
      show y ∈ image f X from
        Exists.intro x (And.intro h2.left.right h2.right)
      done
    done

theorem Theorem_5_5_2_2 {A B : Type} (f : A → B) (W X : Set A)
    (h1 : one_to_one f) : image f (W ∩ X) = image f W ∩ image f X := by
  apply Set.ext
  fix y : B        --Goal: y ∈ image f (W ∩ X) ↔ y ∈ image f W ∩ image f X
  apply Iff.intro
  · -- (→)
    assume h2 : y ∈ image f (W ∩ X)
    show y ∈ image f W ∩ image f X from Theorem_5_5_2_1 f W X h2
    done
  · -- (←)
    assume h2 : y ∈ image f W ∩ image f X  --Goal: y ∈ image f (W ∩ X)
    define at h2                --h2: y ∈ image f W ∧ y ∈ image f X
    rewrite [simp_image, simp_image] at h2
          --h2: (∃ (x : A), x ∈ W ∧ f x = y) ∧ ∃ (x : A), x ∈ X ∧ f x = y
    obtain (x1 : A) (h3 : x1 ∈ W ∧ f x1 = y) from h2.left
    obtain (x2 : A) (h4 : x2 ∈ X ∧ f x2 = y) from h2.right
    have h5 : f x2 = y := h4.right
    rewrite [←h3.right] at h5  --h5: f x2 = f x1
    define at h1                --h1: ∀ (x1 x2 : A), f x1 = f x2 → x1 = x2
    have h6 : x2 = x1 := h1 x2 x1 h5
    rewrite [h6] at h4          --h4: x1 ∈ X ∧ f x1 = y
    show y ∈ image f (W ∩ X) from
      Exists.intro x1 (And.intro (And.intro h3.left h4.left) h3.right)
    done
  done
```

The rest of Section 5.5 of *HTPI* consists of statements for you to try to prove. Here are the statements, written as examples in Lean. Some are correct and some are not; some can be made correct by adding additional hypotheses or weakening the conclusion. Prove as much as you can.

```
--Warning!  Not all of these examples are correct!

example {A B : Type} (f : A → B) (W X : Set A) :
    image f (W ∪ X) = image f W ∪ image f X := sorry
```

```
example {A B : Type} (f : A → B) (W X : Set A) :
    image f (W \ X) = image f W \ image f X := sorry

example {A B : Type} (f : A → B) (W X : Set A) :
    W ⊆ X ↔ image f W ⊆ image f X := sorry

example {A B : Type} (f : A → B) (Y Z : Set B) :
    inverse_image f  (Y ∩ Z) =
        inverse_image f Y ∩ inverse_image f Z := sorry

example {A B : Type} (f : A → B) (Y Z : Set B) :
    inverse_image f  (Y ∪ Z) =
        inverse_image f Y ∪ inverse_image f Z := sorry

example {A B : Type} (f : A → B) (Y Z : Set B) :
    inverse_image f  (Y \ Z) =
        inverse_image f Y \ inverse_image f Z := sorry

example {A B : Type} (f : A → B) (Y Z : Set B) :
    Y ⊆ Z ↔ inverse_image f Y ⊆ inverse_image f Z := sorry

example {A B : Type} (f : A → B) (X : Set A) :
    inverse_image f (image f X) = X := sorry

example {A B : Type} (f : A → B) (Y : Set B) :
    image f (inverse_image f Y) = Y := sorry

example {A : Type} (f : A → A) (C : Set A) :
    closed f C → image f C ⊆ C := sorry

example {A : Type} (f : A → A) (C : Set A) :
    image f C ⊆ C → C ⊆ inverse_image f C := sorry

example {A : Type} (f : A → A) (C : Set A) :
    C ⊆ inverse_image f C → closed f C := sorry

example {A B : Type} (f : A → B) (g : B → A) (Y : Set B)
    (h1 : f ∘ g = id) (h2 : g ∘ f = id) :
    inverse_image f Y = image g Y := sorry
```

6 Mathematical Induction

6.1. Proof by Mathematical Induction

Section 6.1 of *HTPI* introduces a new proof technique called *mathematical induction*. It is used for proving statements of the form `∀ (n : Nat), P n`. Here is how it works:

To prove a goal of the form `∀ (n : Nat), P n`:

First prove `P 0`, and then prove `∀ (n : Nat), P n → P (n + 1)`. The first of these proofs is sometimes called the *base case* and the second the *induction step*.

For an explanation of why this strategy works to establish the truth of `∀ (n : Nat), P n`, see *HTPI*. Here we focus on using mathematical induction in Lean.

To use mathematical induction in a Lean proof, we will use the tactic `by_induc`. If the goal has the form `∀ (n : Nat), P n`, then the `by_induc` tactic leaves the list of givens unchanged, but it replaces the goal with the goals for the base case and induction step. Thus, the effect of the tactic can be summarized as follows:

Tactic State Before Using Strategy

```
    ⋮
⊢ ∀ (n : Nat), P n
```

Tactic State After Using Strategy

```
case Base_Case
  ⋮
⊢ P 0
case Induction_Step
  ⋮
⊢ ∀ (n : Nat), P n → P (n + 1)
```

To illustrate proof by mathematical induction in Lean, we turn first to Example 6.1.2 in *HTPI*, which gives a proof of the statement $\forall n \in \mathbb{N}(3 \mid (n^3 - n))$. For reasons that we will explain a little later, we will prove a slightly different theorem: $\forall n \in \mathbb{N}(3 \mid (n^3 + 2n))$. Here is a proof of the theorem, modeled on the proof in Example 6.1.2 of *HTPI*.

Theorem. *For every natural number n, $3 \mid (n^3 + 2n)$.*

Proof. We use mathematical induction.

Base Case: If $n = 0$, then $n^3 + 2n = 0 = 3 \cdot 0$, so $3 \mid (n^3 + 2n)$.

Induction Step: Let n be an arbitrary natural number and suppose $3 \mid (n^3 + 2n)$. Then we can choose an integer k such that $3k = n^3 + 2n$. Thus,

$$\begin{aligned}
(n+1)^3 + 2(n+1) &= n^3 + 3n^2 + 3n + 1 + 2n + 2 \\
&= (n^3 + 2n) + 3n^2 + 3n + 3 \\
&= 3k + 3n^2 + 3n + 3 \\
&= 3(k + n^2 + n + 1).
\end{aligned}$$

Therefore $3 \mid ((n+1)^3 + 2(n+1))$, as required. $\qquad\square$

Now let's try writing the same proof in Lean. We start, of course, with the `by_induc` tactic.

Lean File	Tactic State in Infoview

```
theorem Like_Example_6_1_2 :
    ∀ (n : Nat), 3 | n ^ 3 + 2 * n := by
  by_induc
  done
```

```
case Base_Case
⊢ 3 | 0 ^ 3 + 2 * 0
case Induction_Step
⊢ ∀ (n : ℕ),
    3 | n ^ 3 + 2 * n →
    3 | (n + 1) ^ 3 +
        2 * (n + 1)
```

The base case is easy: The `define` tactic tells us that the goal means `∃ (c : Nat), 0 ^ 3 + 2 * 0 = 3 * c`, and then `apply Exists.intro 0` changes the goal to `0 ^ 3 + 2 * 0 = 3 * 0`. Both sides are definitionally equal to `0`, so `rfl` finishes off the base case. For the induction step, we begin, as in the *HTPI* proof, by introducing an arbitrary natural number `n` and assuming `3 | n ^ 3 + 2 * n`. This assumption is called the *inductive hypothesis*, so in the Lean proof we give it the identifier `ih`. Our goal now is to prove `3 | (n + 1) ^ 3 + 2 * (n + 1)`.

Lean File

Tactic State in Infoview

```
theorem Like_Example_6_1_2 :
    ∀ (n : Nat), 3 | n ^ 3 + 2 * n := by
  by_induc
  · -- Base Case
    define
    apply Exists.intro 0
    rfl
    done
  · -- Induction Step
    fix n : Nat
    assume ih : 3 | n ^ 3 + 2 * n
    done
  done
```

```
case Induction_Step
n : ℕ
ih : 3 | n ^ 3 + 2 * n
⊢ 3 | (n + 1) ^ 3 +
      2 * (n + 1)
```

The rest of the Lean proof follows the model of the *HTPI* proof: we use the inductive hypothesis to introduce a k such that n ^ 3 + 2 * n = 3 * k, and then we use a calculational proof to show that (n + 1) ^ 3 + 2 * (n + 1) = 3 * (k + n ^ 2 + n + 1).

```
theorem Like_Example_6_1_2 :
    ∀ (n : Nat), 3 | n ^ 3 + 2 * n := by
  by_induc
  · -- Base Case
    define           --Goal: ∃ (c : ℕ), 0 ^ 3 + 2 * 0 = 3 * c
    apply Exists.intro 0
    rfl
    done
  · -- Induction Step
    fix n : Nat
    assume ih : 3 | n ^ 3 + 2 * n
    define at ih    --ih: ∃ (c : ℕ), n ^ 3 + 2 * n = 3 * c
    obtain (k : Nat) (h1 : n ^ 3 + 2 * n = 3 * k) from ih
    define          --Goal: ∃ (c : ℕ), (n + 1) ^ 3 + 2 * (n + 1) = 3 * c
    apply Exists.intro (k + n ^ 2 + n + 1)
    show (n + 1) ^ 3 + 2 * (n + 1) = 3 * (k + n ^ 2 + n + 1) from
      calc (n + 1) ^ 3 + 2 * (n + 1)
          = n ^ 3 + 2 * n + 3 * n ^ 2 + 3 * n + 3 := by ring
        _ = 3 * k + 3 * n ^ 2 + 3 * n + 3 := by rw [h1]
        _ = 3 * (k + n ^ 2 + n + 1) := by ring
    done
  done
```

Next we'll look at Example 6.1.1 in *HTPI*, which proves that for every natural number n, $2^0 + 2^1 + \cdots + 2^n = 2^{n+1} - 1$. Once again, we will change the theorem slightly before proving it in Lean. What we will prove is that for every n, $(2^0 + 2^1 + \cdots + 2^n) + 1 = 2^{n+1}$. To understand this theorem you must be able to recognize what the "$\cdots$" stands for. A human reader will probably realize that the numbers being added up here are the numbers of the form 2^i, where i runs through all of the natural numbers from 0 to n. But Lean can't be expected to figure out this pattern, so we must be more explicit.

Section 6.3 of *HTPI* introduces the explicit notation that mathematicians usually use for such sums. If f is a function whose domain is the natural numbers, then

$$\sum_{i=0}^{n} f(i) = f(0) + f(1) + \cdots + f(n).$$

More generally, if $k \leq n$ then

$$\sum_{i=k}^{n} f(i) = f(k) + f(k+1) + \cdots + f(n).$$

The notation we will use in Lean for this sum is `Sum i from k to n, f i`. Thus, a mathematician would state our theorem like this:

Theorem. *For every natural number n,*

$$\left(\sum_{i=0}^{n} 2^i \right) + 1 = 2^{n+1}.$$

And to state the same theorem in Lean, we will write:

```
theorem Like_Example_6_1_1 :
    ∀ (n : Nat), (Sum i from 0 to n, 2 ^ i) + 1 = 2 ^ (n + 1)
```

We will have more to say later about how the notation `Sum i from k to n, f i` is defined. But to use the notation in a proof, we will just need to know a few theorems. The `#check` command will tell us the meanings of the theorems `sum_base`, `sum_step`, and `sum_from_zero_step`:

```
@sum_base : ∀ {A : Type} [inst : AddZeroClass A] {k : ℕ} {f : ℕ → A},
            Sum i from k to k, f i = f k

@sum_step : ∀ {A : Type} [inst : AddZeroClass A] {k n : ℕ} {f : ℕ → A},
            k ≤ n → Sum i from k to n + 1, f i =
              (Sum i from k to n, f i) + f (n + 1)
```

```
@sum_from_zero_step :
            ∀ {A : Type} [inst : AddZeroClass A] {n : ℕ} {f : ℕ → A},
            Sum i from 0 to n + 1, f i =
              (Sum i from 0 to n, f i) + f (n + 1)
```

As usual, we don't need to pay too much attention to the implicit arguments in the first line of each statement. What is important is that `sum_base` can be used to prove any statement of the form

```
Sum i from k to k, f i = f k
```

and `sum_step` proves any statement of the form

```
k ≤ n → Sum i from k to n + 1, f i = (Sum i from k to n, f i) + f (n + 1)
```

In the case `k = 0`, we have the simpler theorem `sum_from_zero_step`, which proves

```
Sum i from 0 to n + 1, f i = (Sum i from 0 to n, f i) + f (n + 1)
```

With that preparation, we can start on the proof. Once again we begin with the `by_induc` tactic. Our goal for the base case is (`Sum i from 0 to 0, 2 ^ i) + 1 = 2 ^ (0 + 1)`. To deal with the term `Sum i from 0 to 0, 2 ^ i`, we use that fact that `sum_base` proves `Sum i from 0 to 0, 2 ^ i = 2 ^ 0`. It follows that the tactic `rewrite [sum_base]` will change the goal to `2 ^ 0 + 1 = 2 ^ (0 + 1)`. Of course, this means `2 = 2`, so `rfl` finishes the base case. For the induction step, we start by introducing an arbitrary natural number `n` and assuming the inductive hypothesis.

```
theorem Like_Example_6_1_1 :
    ∀ (n : Nat), (Sum i from 0 to n, 2 ^ i) + 1 = 2 ^ (n + 1) := by
  by_induc
  · -- Base Case
    rewrite [sum_base]
    rfl
    done
  · -- Induction Step
    fix n : Nat
    assume ih : (Sum i from 0 to n, 2 ^ i) + 1 = 2 ^ (n + 1)
    done
  done
```

Our goal is now (`Sum i from 0 to n + 1, 2 ^ i) + 1 = 2 ^ (n + 1 + 1)`, and we use a calculational proof to prove this. Often the key to the proof of the induction step is to find a relationship between the inductive hypothesis and the goal. In this case, that means finding a relationship between `Sum i from 0 to n, 2 ^ i` and `Sum i from 0 to n + 1, 2 ^ i`. The relationship we need is given by the theorem `sum_from_zero_step`. The tactic `rewrite [sum_from_zero_step]`

will replace `Sum i from 0 to n + 1, 2 ^ i` with `(Sum i from 0 to n, 2 ^ i) + 2 ^ (n + 1)`. The rest of the calculation proof involves straightforward algebra, handled by the `ring` tactic, together with an application of the inductive hypothesis.

```
theorem Like_Example_6_1_1 :
    ∀ (n : Nat), (Sum i from 0 to n, 2 ^ i) + 1 = 2 ^ (n + 1) := by
  by_induc
  · -- Base Case
    rewrite [sum_base]
    rfl
    done
  · -- Induction Step
    fix n : Nat
    assume ih : (Sum i from 0 to n, 2 ^ i) + 1 = 2 ^ (n + 1)
    show (Sum i from 0 to n + 1, 2 ^ i) + 1 = 2 ^ (n + 1 + 1) from
      calc (Sum i from 0 to n + 1, 2 ^ i) + 1
          = (Sum i from 0 to n, 2 ^ i) + 2 ^ (n + 1) + 1 :=
            by rw [sum_from_zero_step]
        _ = (Sum i from 0 to n, 2 ^ i) + 1 + 2 ^ (n + 1) := by ring
        _ = 2 ^ (n + 1) + 2 ^ (n + 1) := by rw [ih]
        _ = 2 ^ (n + 1 + 1) := by ring
    done
  done
```

The last example in Section 6.1 of *HTPI* gives a proof of the statement $\forall n \geq 5(2^n > n^2)$. The proof is by mathematical induction, but since we are only interested in natural numbers greater than or equal to 5, it uses 5 in the base case instead of 0. Here are the theorem and proof from *HTPI*:

Theorem. *For every natural number $n \geq 5$, $2^n > n^2$.*

Proof. By mathematical induction.

Base case: When $n = 5$ we have $2^n = 32 > 25 = n^2$.

Induction step: Let $n \geq 5$ be arbitrary, and suppose that $2^n > n^2$. Then

$$
\begin{aligned}
2^{n+1} &= 2 \cdot 2^n \\
&> 2n^2 && \text{(inductive hypothesis)} \\
&= n^2 + n^2 \\
&\geq n^2 + 5n && \text{(since } n \geq 5\text{)} \\
&= n^2 + 2n + 3n \\
&> n^2 + 2n + 1 = (n+1)^2. && \square
\end{aligned}
$$

Notice that the sequence of calculations at the end of the proof mixes $=$, $>$, and $\geq$ in a way that establishes the final conclusion $2^{n+1} > (n+1)^2$. As we'll see, such a mixture is allowed in calculational proofs in Lean as well.

To write this proof in Lean, there is no need to specify that the base case should be `n = 5`; the `by_induc` tactic is smart enough to figure that out on its own, as you can see in the tactic state below:

Lean File

```
theorem Example_6_1_3 :
    ∀ n ≥ 5, 2 ^ n > n ^ 2 := by
  by_induc
  done
```

Tactic State in Infoview

```
case Base_Case
⊢ 2 ^ 5 > 5 ^ 2
case Induction_Step
⊢ ∀ (n : ℕ), n ≥ 5 →
  2 ^ n > n ^ 2 →
  2 ^ (n + 1) >
      (n + 1) ^ 2
```

To complete this proof we'll use two tactics we haven't used before: `norm_num` and `linarith`. The base case can be proven by simply doing the arithmetic necessary to check that the inequality holds when `n = 5`. The tactic `norm_num` does such numerical calculations, and it proves the base case.

For the induction step, we introduce an arbitrary natural number n, assume `n ≥ 5`, and assume the inductive hypothesis, `2 ^ n > n ^ 2`. Then we use a calculational proof to imitate the reasoning at the end of the *HTPI* proof. The tactic `linarith` makes inferences that involve combining linear equations and inequalities. It is able to prove almost all of the inequalities in the calculational proof. The exception is `n * n ≥ 5 * n` (which is not linear because of the term `n * n`). So we prove that inequality separately, using a theorem from Lean's library, `Nat.mul_le_mul_right`. The command `#check @Nat.mul_le_mul_right` tells us the meaning of that theorem:

```
@Nat.mul_le_mul_right : ∀ {n m : ℕ} (k : ℕ), n ≤ m → n * k ≤ m * k
```

Thus, `Nat.mul_le_mul_right n` can be used to prove the statement `5 ≤ n → 5 * n ≤ n * n`. Lean recognizes `x ≥ y` as meaning the same thing as `y ≤ x`, so we can apply this statement to our assumption `n ≥ 5` to prove that `n * n ≥ 5 * n`. Once we have proven that inequality, the `linarith` tactic can use it to complete the required inequality reasoning.

```
theorem Example_6_1_3 : ∀ n ≥ 5, 2 ^ n > n ^ 2 := by
  by_induc
  · -- Base Case
    norm_num
    done
```

```
· -- Induction Step
  fix n : Nat
  assume h1 : n ≥ 5
  assume ih : 2 ^ n > n ^ 2
  have h2 : n * n ≥ 5 * n := Nat.mul_le_mul_right n h1
  show 2 ^ (n + 1) > (n + 1) ^ 2 from
    calc 2 ^ (n + 1)
        = 2 * 2 ^ n := by ring
      _ > n ^ 2 + 2 * n + 1 := by linarith
      _ = (n + 1) ^ 2 := by ring
  done
done
```

Finally, we turn to the question of why we made small changes in two of the examples from *HTPI*. Perhaps you have guessed by now that we were trying to avoid the use of subtraction. All of the numbers in the examples in this section were natural numbers, and subtraction of natural numbers is problematic. In the natural numbers, `3 - 2` is equal to `1`, but what is `2 - 3`? Lean's answer is `0`.

```
#eval 2 - 3     --Answer: 0
```

In Lean, if `a` and `b` are natural numbers and `a < b`, then `a - b` is defined to be `0`. As a result, the algebraic laws of natural number subtraction are complicated. For example, `2 - 3 + 1 = 0 + 1 = 1`, but `2 + 1 - 3 = 3 - 3 = 0`, so it is not true that for all natural numbers `a`, `b`, and `c`, `a - b + c = a + c - b`.

If you thought that the answer to the subtraction problem `2 - 3` was `-1`, then you automatically switched from the natural numbers to the integers. (Recall that the natural numbers are the numbers 0, 1, 2, …, while the integers are the numbers …, –3, –2, –1, 0, 1, 2, 3, ….) To a human mathematician, this is a perfectly natural thing to do: the natural numbers are a subset of the integers, so `2` and `3` are not only natural numbers but also integers, and we can compute `2 - 3` in the integers.

However, that's not how things work in Lean. In Lean, different types are completely separate. In particular, `Nat` and `Int` are separate types, and therefore the natural numbers are not a subset of the integers. Of course, there is an integer 2, but it is different from the natural number 2. By default, Lean assumes that `2` denotes the natural number 2, but you can specify that you want the integer 2 by writing `(2 : Int)`. Subtraction of integers in Lean is the subtraction you are familiar with, and it has all the algebraic properties you would expect. If we want to use subtraction in the theorems in this section, we are better off using familiar integer subtraction rather than funky natural number subtraction.

To prove the theorem in Example 6.1.1 as it appears in *HTPI*, we could state the theorem like this:

```
theorem Example_6_1_1 :
    ∀ (n : Nat), Sum i from 0 to n, (2 : Int) ^ i =
    (2 : Int) ^ (n + 1) - (1 : Int)
```

The expression `Sum i from 0 to n, (2 : Int) ^ i` denotes a sum of integers, so it is an integer. Similarly, the right side of the equation is an integer, and the equation asserts the equality of two integers. The subtraction on the right side of the equation is integer subtraction, so we can use the usual algebraic laws to reason about it. In fact, the proof of the theorem in this form is not hard:

```
theorem Example_6_1_1 :
    ∀ (n : Nat), Sum i from 0 to n, (2 : Int) ^ i =
    (2 : Int) ^ (n + 1) - (1 : Int) := by
  by_induc
  · -- Base Case
    rewrite [sum_base]
    rfl
    done
  · -- Induction Step
    fix n : Nat
    assume ih : Sum i from 0 to n, (2 : Int) ^ i =
      (2 : Int) ^ (n + 1) - (1 : Int)
    rewrite [sum_from_zero_step, ih]
      --Goal: 2 ^ (n + 1) - 1 + 2 ^ (n + 1) = 2 ^ (n + 1 + 1) - 1
    ring
    done
  done
```

If you change `(2 : Int)` and `(1 : Int)` to `2` and `1`, then the right side of the equation will be a difference of two natural numbers, and Lean will interpret the subtraction as natural number subtraction. The proof won't work because the `ring` tactic is not able to deal with the peculiar algebraic properties of natural number subtraction. (The theorem is still true, but the proof is harder.)

Exercises

1.
```
theorem Like_Exercise_6_1_1 :
    ∀ (n : Nat), 2 * Sum i from 0 to n, i = n * (n + 1) := sorry
```

2.
```
theorem Like_Exercise_6_1_4 :
    ∀ (n : Nat), Sum i from 0 to n, 2 * i + 1 = (n + 1) ^ 2 := sorry
```

3.
```
theorem Exercise_6_1_9a : ∀ (n : Nat), 2 | n ^ 2 + n := sorry
```

4.
```
theorem Exercise_6_1_13 :
    ∀ (a b : Int) (n : Nat), (a - b) | (a ^ n - b ^ n) := sorry
```

5.
```
theorem Exercise_6_1_15 : ∀ n ≥ 10, 2 ^ n > n ^ 3 := sorry
```

For the next two exercises you will need the following definitions:

```
def nat_even (n : Nat) : Prop := ∃ (k : Nat), n = 2 * k

def nat_odd (n : Nat) : Prop := ∃ (k : Nat), n = 2 * k + 1
```

6.
```
theorem Exercise_6_1_16a1 :
    ∀ (n : Nat), nat_even n ∨ nat_odd n := sorry
```

7.
```
theorem Exercise_6_1_16a2 :
    ∀ (n : Nat), ¬(nat_even n ∧ nat_odd n) := sorry
```

6.2. More Examples

We saw in the last section that mathematical induction can be used to prove theorems about calculations involving natural numbers. But mathematical induction has a much wider range of uses. Section 6.2 of *HTPI* illustrates this by proving two theorems about finite sets.

How can mathematical induction be used to prove a statement about finite sets? To say that a set is finite means that it has n elements, for some natural number n. Thus, to say that all finite sets have some property, we can say that for every natural number n, every set with n elements has the property. Since this statement starts with "for every natural number n," we can use mathematical induction to try to prove it.

What does it mean to say that a set "has n elements"? Section 6.2 of *HTPI* says that for the proofs in that section, "an intuitive understanding of this concept will suffice." Unfortunately, intuition is not Lean's strong suit! So we'll need to be more explicit about how to talk about finite sets in Lean.

In Chapter 8, we'll define `numElts X n` to be a proposition saying that the set `X` has `n` elements, and we'll prove several theorems involving that proposition. Those theorems make precise and explicit the intuitive ideas that we'll need in this section. We'll state those theorems here and use them in our proofs, but you'll have to wait until Chapter 8 to see how they are proven. Here are the theorems we'll need:

```
theorem zero_elts_iff_empty {A : Type} (X : Set A) :
    numElts X 0 ↔ empty X

theorem one_elt_iff_singleton {A : Type} (X : Set A) :
    numElts X 1 ↔ ∃ (x : A), X = {x}

theorem nonempty_of_pos_numElts {A : Type} {X : Set A} {n : Nat}
    (h1 : numElts X n) (h2 : n > 0) : ∃ (x : A), x ∈ X

theorem remove_one_numElts {A : Type} {X : Set A} {n : Nat} {x : A}
    (h1 : numElts X (n + 1)) (h2 : x ∈ X) : numElts (X \ {x}) n
```

These theorems should make intuitive sense. The first says that a set has zero elements if and only if it is empty, and the second says that a set has one element if and only if it is a singleton set. The third theorem says that if a set has a positive number of elements, then there is something in the set. And the fourth says that if a set has $n + 1$ elements and you remove one element, then the resulting set has n elements. You can probably guess that we'll be using the last theorem in the induction steps of our proofs.

Our first theorem about finite sets says that if R is a partial order on A, then every finite, nonempty subset of A has an R-minimal element. (This is not true in general for infinite subsets of A. Can you think of an example of an infinite subset of a partially ordered set that has no minimal element?) To say that a set is finite and nonempty we can say that it has n elements for some $n \geq 1$. So here's how we state our theorem in Lean:

```
theorem Example_6_2_1 {A : Type} (R : BinRel A) (h : partial_order R) :
    ∀ n ≥ 1, ∀ (B : Set A), numElts B n →
      ∃ (x : A), minimalElt R x B
```

When we use mathematical induction to prove this theorem, the base case will be `n = 1`. To write the proof for the base case, we start by assuming `B` is a set with one element. We can then use the theorem `one_elt_iff_singleton` to conclude that `B = {b}`, for some b of type `A`. We need to prove that `B` has a minimal element, and the only possibility for the minimal element is b. Verifying that `minimalElt R b B` is straightforward. Here is the proof of the base case:

```
theorem Example_6_2_1 {A : Type} (R : BinRel A) (h : partial_order R) :
    ∀ n ≥ 1, ∀ (B : Set A), numElts B n →
      ∃ (x : A), minimalElt R x B := by
  by_induc
  · -- Base Case
    fix B : Set A
    assume h2 : numElts B 1
    rewrite [one_elt_iff_singleton] at h2
    obtain (b : A) (h3 : B = {b}) from h2
    apply Exists.intro b
    define            --Goal: b ∈ B ∧ ¬∃ (x : A), x ∈ B ∧ R x b ∧ x ≠ b
    apply And.intro
    · -- Proof that b ∈ B
      rewrite [h3]    --Goal: b ∈ {b}
      define
      rfl
      done
    · -- Proof that nothing in B is smaller than b
      by_contra h4
      obtain (x : A) (h5 : x ∈ B ∧ R x b ∧ x ≠ b) from h4
      have h6 : x ∈ B := h5.left
      rewrite [h3] at h6    --h6: x ∈ {b}
      define at h6          --h6: x = b
      show False from h5.right.right h6
      done
    done
  · -- Induction Step

    done
  done
```

Notice that since the definition of `minimalElt R b B` involves a negative statement, we found it convenient to use proof by contradiction to prove it.

For the induction step, we assume that `n ≥ 1` and that every set with `n` elements has an `R`-minimal element. We must prove that every set with `n + 1` elements has a minimal element, so we let `B` be an arbitrary set with `n + 1` elements. To apply the inductive hypothesis, we need a set with `n` elements. So we pick some `b ∈ B` (using the theorem `nonempty_of_pos_numElts`) and then remove it from `B` to get the set `B' = B \ {b}`. The theorem `remove_one_numElts` tells us that `B'` has `n` elements, so by the inductive hypothesis, we can then let `c` be a minimal element of `B'`. We now know about two elements of `B`: `b` and `c`. Which will be a minimal element of `B`? As explained in *HTPI*, it depends on whether or not `R b c`. We'll prove that if `R b c`, then `b` is a minimal element of `B`, and if not, then `c` is a minimal element. It will be convenient to

prove these last two facts separately as lemmas. The first lemma says that in the situation at this point in the proof, if R b c, then b is an R-minimal element of B. Here is the proof.

```lean
lemma Lemma_6_2_1_1 {A : Type} {R : BinRel A} {B : Set A} {b c : A}
    (h1 : partial_order R) (h2 : b ∈ B) (h3 : minimalElt R c (B \ {b}))
    (h4 : R b c) : minimalElt R b B := by
  define at h3
    --h3: c ∈ B \ {b} ∧ ¬∃ (x : A), x ∈ B \ {b} ∧ R x c ∧ x ≠ c
  define  --Goal: b ∈ B ∧ ¬∃ (x : A), x ∈ B ∧ R x b ∧ x ≠ b
  apply And.intro h2  --Goal: ¬∃ (x : A), x ∈ B ∧ R x b ∧ x ≠ b
  contradict h3.right with h5
  obtain (x : A) (h6 : x ∈ B ∧ R x b ∧ x ≠ b) from h5
  apply Exists.intro x
  apply And.intro
  · -- Proof that x ∈ B \ {b}
    show x ∈ B \ {b} from And.intro h6.left h6.right.right
    done
  · -- Proof that R x c ∧ x ≠ c
    have Rtrans : transitive R := h1.right.left
    have h7 : R x c := Rtrans x b c h6.right.left h4
    apply And.intro h7
    by_contra h8
    rewrite [h8] at h6
    have Rantisymm : antisymmetric R := h1.right.right
    have h9 : c = b := Rantisymm c b h6.right.left h4
    show False from h6.right.right h9
    done
  done
```

The second lemma says that if ¬R b c, then c is an R-minimal element of B. We'll leave the proof as an exercise for you:

```lean
lemma Lemma_6_2_1_2 {A : Type} {R : BinRel A} {B : Set A} {b c : A}
    (h1 : partial_order R) (h2 : b ∈ B) (h3 : minimalElt R c (B \ {b}))
    (h4 : ¬R b c) : minimalElt R c B := sorry
```

With this preparation, we are finally ready to give the proof of the induction step of Example_6_2_1:

```lean
theorem Example_6_2_1 {A : Type} (R : BinRel A) (h : partial_order R) :
    ∀ n ≥ 1, ∀ (B : Set A), numElts B n →
      ∃ (x : A), minimalElt R x B := by
```

```
  by_induc
  · -- Base Case
    ...
  · -- Induction Step
    fix n : Nat
    assume h2 : n ≥ 1
    assume ih : ∀ (B : Set A), numElts B n → ∃ (x : A), minimalElt R x B
    fix B : Set A
    assume h3 : numElts B (n + 1)
    have h4 : n + 1 > 0 := by linarith
    obtain (b : A) (h5 : b ∈ B) from nonempty_of_pos_numElts h3 h4
    let B' : Set A := B \ {b}
    have h6 : numElts B' n := remove_one_numElts h3 h5
    obtain (c : A) (h7 : minimalElt R c B') from ih B' h6
    by_cases h8 : R b c
    · -- Case 1. h8: R b c
      have h9 : minimalElt R b B := Lemma_6_2_1_1 h h5 h7 h8
      show ∃ (x : A), minimalElt R x B from Exists.intro b h9
      done
    · -- Case 2. h8: ¬R b c
      have h9 : minimalElt R c B := Lemma_6_2_1_2 h h5 h7 h8
      show ∃ (x : A), minimalElt R x B from Exists.intro c h9
      done
    done
  done
```

We'll consider one more theorem from Section 6.2 of *HTPI*. Example 6.2.2 proves that a partial order on a finite set can always be extended to a total order. Rather than give that proof, we are going to prove the more general theorem that is stated in Exercise 2 in Section 6.2 of *HTPI*. To explain the theorem in that exercise, it will be helpful to introduce a bit of terminology. Suppose R is a partial order on A and b has type A. We will say that b is R-*comparable to everything* if ∀ (x : A), R b x ∨ R x b. If B is a set of objects of type A, we say that B is R-*comparable to everything* if every element of B is R-comparable to everything; that is, if ∀ b ∈ B, ∀ (x : A), R b x ∨ R x b. Finally, we say that another binary relation T *extends* R if ∀ (x y : A), R x y → T x y. We are going to prove that if R is a partial order on A and B is a finite set of objects of type A, then there is a partial order T that extends R such that B is T-comparable to everything. In other words, we are going to prove the following theorem:

```
  theorem Exercise_6_2_2 {A : Type} (R : BinRel A) (h : partial_order R) :
      ∀ (n : Nat) (B : Set A), numElts B n → ∃ (T : BinRel A),
      partial_order T ∧ (∀ (x y : A), R x y → T x y) ∧
      ∀ x ∈ B, ∀ (y : A), T x y ∨ T y x
```

In the exercises, we will ask you to show that this implies the theorem in Example 6.2.2.

It will be helpful to begin with a warm-up exercise. We'll show that a partial order can always be extended to make a single object comparable to everthing. In other words, we'll show that if R is a partial order on A and b has type A, then we can define a partial order T extending R such that b is T-comparable to everything. To define T, we will need to make sure that for every x of type A, either T b x or T x b. If R x b, then since T must extend R, we must have T x b. If ¬R x b, then we will define T so that T b x. But notice that if we follow this plan, then for any x and y, if we have R x b and ¬R y b, then we will have T x b and T b y, and since T must be transitive, we must then have T x y. Summing up, if we have R x y then we must have T x y, and if we have R x b and ¬R y b then we will also need to have T x y. So let's try defining T x y to mean R x y ∨ (R x b ∧ ¬R y b).

It will be useful to have a name for this relation T. Since it is an extension of R determined by the element b, we will give it the name extendPO R b. Here is the definition of this relation:

```
def extendPO {A : Type} (R : BinRel A) (b : A) (x y : A) : Prop :=
    R x y ∨ (R x b ∧ ¬R y b)
```

We need to prove a number of things about extendPO R b. First of all, we need to prove that it is a partial order. We'll leave most of the details as exercises for you:

```
lemma extendPO_is_ref {A : Type} (R : BinRel A) (b : A)
    (h : partial_order R) : reflexive (extendPO R b) := sorry

lemma extendPO_is_trans {A : Type} (R : BinRel A) (b : A)
    (h : partial_order R) : transitive (extendPO R b) := sorry

lemma extendPO_is_antisymm {A : Type} (R : BinRel A) (b : A)
    (h : partial_order R) : antisymmetric (extendPO R b) := sorry

lemma extendPO_is_po {A : Type} (R : BinRel A) (b : A)
    (h : partial_order R) : partial_order (extendPO R b) :=
  And.intro (extendPO_is_ref R b h)
    (And.intro (extendPO_is_trans R b h) (extendPO_is_antisymm R b h))
```

It is easy to prove that extendPO R b extends R:

```
lemma extendPO_extends {A : Type} (R : BinRel A) (b : A) (x y : A) :
    R x y → extendPO R b x y := by
  assume h1 : R x y
  define
  show R x y ∨ R x b ∧ ¬R y b from Or.inl h1
```

```
      done
```

Finally, we verify that extendPO R b does what it was supposed to do: it makes b comparable with everything:

```
  lemma extendPO_all_comp {A : Type} (R : BinRel A) (b : A)
      (h : partial_order R) :
      ∀ (x : A), extendPO R b b x ∨ extendPO R b x b := by
    have Rref : reflexive R := h.left
    fix x : A
    or_left with h1
    define at h1     --h1: ¬(R x b ∨ R x b ∧ ¬R b b)
    demorgan at h1   --h1: ¬R x b ∧ ¬(R x b ∧ ¬R b b)
    define           --Goal: R b x ∨ R b b ∧ ¬R x b
    apply Or.inr
    show R b b ∧ ¬R x b from And.intro (Rref b) h1.left
    done
```

With this preparation, we can finally return to our theorem Exercise_6_2_2. We will prove it by mathematical induction. In the base case we must show that if B has 0 elements then we can extend R to make everything in B comparable to everything. Of course, no extension is necessary, since it is vacuously true that all elements of B are R-comparable to everything. For the induction step, after assuming the inductive hypothesis, we must prove that if B has n + 1 elements then we can extend R to make all elements of B comparable to everything. As before, we choose b ∈ B and let B' = B \ {b}. By inductive hypothesis, we can find an extension T' of R that makes all elements of B' comparable to everything, so we just have to extend T' further to make b comparable to everything. But as we have just seen, we can do this with extendPO T' b.

```
  theorem Exercise_6_2_2 {A : Type} (R : BinRel A) (h : partial_order R) :
      ∀ (n : Nat) (B : Set A), numElts B n → ∃ (T : BinRel A),
      partial_order T ∧ (∀ (x y : A), R x y → T x y) ∧
      ∀ x ∈ B, ∀ (y : A), T x y ∨ T y x := by
    by_induc
    · -- Base Case
      fix B : Set A
      assume h2 : numElts B 0
      rewrite [zero_elts_iff_empty] at h2
      define at h2     --h2 : ¬∃ (x : A), x ∈ B
      apply Exists.intro R
      apply And.intro h
      apply And.intro
```

```
· -- Proof that R extends R
  fix x : A; fix y : A
  assume h3 : R x y
  show R x y from h3
  done
· -- Proof that everything in B comparable to everything under R
  fix x : A
  assume h3 : x ∈ B
  contradict h2
  show ∃ (x : A), x ∈ B from Exists.intro x h3
  done
done
· -- Induction Step
  fix n : Nat
  assume ih : ∀ (B : Set A), numElts B n → ∃ (T : BinRel A),
    partial_order T ∧ (∀ (x y : A), R x y → T x y) ∧
    ∀ (x : A), x ∈ B → ∀ (y : A), T x y ∨ T y x
  fix B
  assume h2 : numElts B (n + 1)
  have h3 : n + 1 > 0 := by linarith
  obtain (b : A) (h5 : b ∈ B) from nonempty_of_pos_numElts h2 h3
  let B' : Set A := B \ {b}
  have h6 : numElts B' n := remove_one_numElts h2 h5
  have h7 : ∃ (T : BinRel A), partial_order T ∧
    (∀ (x y : A), R x y → T x y) ∧
    ∀ (x : A), x ∈ B' → ∀ (y : A), T x y ∨ T y x := ih B' h6
  obtain (T' : BinRel A)
    (h8 : partial_order T' ∧ (∀ (x y : A), R x y → T' x y) ∧
    ∀ (x : A), x ∈ B' → ∀ (y : A), T' x y ∨ T' y x) from h7
  have h9 : partial_order T' := h8.left
  let T : BinRel A := extendPO T' b
  apply Exists.intro T
  apply And.intro (extendPO_is_po T' b h9)
  apply And.intro
  · -- Proof that T extends R
    fix x : A; fix y : A
    assume h10 : R x y
    have h11 : T' x y := h8.right.left x y h10
    show T x y from (extendPO_extends T' b x y h11)
    done
  · -- Proof that everything in B comparable to everything under T
    fix x : A
```

```
        assume h10 : x ∈ B
        by_cases h11 : x = b
        · -- Case 1. h11: x = b
          rewrite [h11]
          show ∀ (y : A), T b y ∨ T y b from extendPO_all_comp T' b h9
          done
        · -- Case 2. h11: x ≠ b
          have h12 : x ∈ B' := And.intro h10 h11
          fix y : A
          have h13 : T' x y ∨ T' y x := h8.right.right x h12 y
          by_cases on h13
          · -- Case 1. h13: T' x y
            show T x y ∨ T y x from
              Or.inl (extendPO_extends T' b x y h13)
            done
          · -- Case 2. h13: T' y x
            show T x y ∨ T y x from
              Or.inr (extendPO_extends T' b y x h13)
            done
          done
        done
      done
    done
  done
```

Exercises

1.
```
lemma Lemma_6_2_1_2 {A : Type} {R : BinRel A} {B : Set A} {b c : A}
    (h1 : partial_order R) (h2 : b ∈ B) (h3 : minimalElt R c (B \ {b}))
    (h4 : ¬R b c) : minimalElt R c B := sorry
```

2.
```
lemma extendPO_is_ref {A : Type} (R : BinRel A) (b : A)
    (h : partial_order R) : reflexive (extendPO R b) := sorry
```

3.
```
lemma extendPO_is_trans {A : Type} (R : BinRel A) (b : A)
    (h : partial_order R) : transitive (extendPO R b) := sorry
```

4.
```
lemma extendPO_is_antisymm {A : Type} (R : BinRel A) (b : A)
    (h : partial_order R) : antisymmetric (extendPO R b) := sorry
```

5.
```
theorem Exercise_6_2_3 (A : Type) (R : BinRel A)
    (h : total_order R) : ∀ n ≥ 1, ∀ (B : Set A),
    numElts B n → ∃ (b : A), smallestElt R b B := sorry
```

6.
```
theorem Exercise_6_2_4a {A : Type} (R : BinRel A)
    (h : ∀ (x y : A), R x y ∨ R y x) : ∀ n ≥ 1, ∀ (B : Set A),
    numElts B n → ∃ x ∈ B, ∀ y ∈ B, ∃ (z : A), R x z ∧ R z y := sorry
--Hint:  First prove that R is reflexive
```

7.
```
theorem Like_Exercise_6_2_16 (f : Nat → Nat) (h : one_to_one f) :
    ∀ (n : Nat) (A : Set Nat), numElts A n →
    closed f A → ∀ y ∈ A, ∃ x ∈ A, f x = y := sorry
```

6.3. Recursion

In the last two sections, we saw that we can prove that all natural numbers have some property by proving that 0 has the property, and also that for every natural number n, if n has the property then so does $n+1$. In this section we will see that a similar idea can be used to define a function whose domain is the natural numbers. We can define a function f with domain $\mathbb{N}$ by specifying the value of $f(0)$, and also saying how to compute $f(n+1)$ if you already know the value of $f(n)$.

For example, we can define a function $f : \mathbb{N} \to \mathbb{N}$ as follows:

$f(0) = 1$;
for every $n \in \mathbb{N}$, $f(n+1) = (n+1) \cdot f(n)$.

Here is the same definition written in Lean. (For reasons that will become clear shortly, we have given the function the name `fact`.)

```
def fact (x : Nat) : Nat :=
  match x with
    | 0 => 1
    | n + 1 => (n + 1) * fact n
```

Lean can use this definition to compute `fact x` for any natural number `x`. The `match` statement tells Lean to try to match the input `x` with one of the two patterns `0` and `n + 1`, and then to use the corresponding formula after `=>` to compute `fact x`. For example, if we ask Lean for `fact 4`, it first checks if `4` matches `0`. Since it doesn't, it goes on to the next line and determines that `4` matches the pattern `n + 1`, with `n = 3`, so it uses the formula `fact 4 = 4 * fact 3`. Of

course, now it must compute `fact 3`, which it does in the same way: `3` matches `n + 1` with `n = 2`, so `fact 3 = 3 * fact 2`. Continuing in this way, Lean determines that

```
fact 4 = 4 * fact 3 = 4 * (3 * fact 2) = 4 * (3 * (2 * fact 1))
       = 4 * (3 * (2 * (1 * fact 0))) = 4 * (3 * (2 * (1 * 1))) = 24
```

You can confirm this with the `#eval` command:

```
#eval fact 4    --Answer: 24
```

Of course, by now you have probably guessed why we used the name `fact` for his function: `fact x` is x factorial—the product of all the numbers from 1 to x.

This style of definition is called a *recursive* definition. If a function is defined by a recursive definition, then theorems about that function are often most easily proven by induction. For example, here is a theorem about the factorial function. It is Example 6.3.1 in *HTPI*, and we begin the Lean proof by imitating the proof in *HTPI*.

```
theorem Example_6_3_1 : ∀ n ≥ 4, fact n > 2 ^ n := by
  by_induc
  · -- Base Case
    norm_num
    done
  · -- Induction Step
    fix n : Nat
    assume h1 : n ≥ 4
    assume ih : fact n > 2 ^ n
    show fact (n + 1) > 2 ^ (n + 1) from
      calc fact (n + 1)
          = (n + 1) * fact n := by rfl
        _ > (n + 1) * 2 ^ n := sorry
        _ > 2 * 2 ^ n := sorry
        _ = 2 ^ (n + 1) := by ring
    done
  done
```

There are two steps in the calculational proof at the end that require justification. The first says that `(n + 1) * fact n > (n + 1) * 2 ^ n`, which should follow from the inductive hypothesis `ih : fact n > 2 ^ n` by multiplying both sides by `n + 1`. Is there a theorem that would justify this inference?

This may remind you a step in `Example_6_1_3` where we used the theorem `Nat.mul_le_mul_right`, which says `∀ {n m : ℕ} (k : ℕ), n ≤ m → n * k ≤ m * k`. Our situation in this example is similar, but it involves a strict inequality (`>` rather than `≥`) and it involves multiplying on the left rather than the right. Many theorems about inequalities in Lean's library contain

either `le` (for "less than or equal to") or `lt` (for "less than") in their names, but they can also be used to prove statements involving `≥` or `>`. Perhaps the theorem we need is named something like `Nat.mul_lt_mul_left`. If you type `#check @Nat.mul_lt_mul` into VS Code, a pop-up window will appear listing several theorems that begin with `Nat.mul_lt_mul`. There is no `Nat.mul_lt_mul_left`, but there is a theorem called `Nat.mul_lt_mul_of_pos_left`, and its meaning is

```
@Nat.mul_lt_mul_of_pos_left : ∀ {n m k : ℕ},
                                  n < m → k > 0 → k * n < k * m
```

Lean has correctly reminded us that, to multiply both sides of a strict inequality by a number `k`, we need to know that `k > 0`. So in our case, we'll need to prove that `n + 1 > 0`. Once we have that, we can use the theorem `Nat.mul_lt_mul_of_pos_left` to eliminate the first `sorry`.

The second `sorry` is similar: `(n + 1) * 2 ^ n > 2 * 2 ^ n` should follow from `n + 1 > 2` and `2 ^ n > 0`, and you can verify that the theorem that will justify this inference is `Nat.mul_lt_mul_of_pos_right`.

So we have three inequalities that we need to prove before we can justify the steps of the calculational proof: `n + 1 > 0`, `n + 1 > 2`, and `2 ^ n > 0`. We'll insert `have` steps before the calculational proof to assert these three inequalities. If you try it, you'll find that `linarith` can prove the first two, but not the third.

How can we prove `2 ^ n > 0`? It is often helpful to think about whether there is a general principle that is behind a statement we are trying to prove. In our case, the inequality `2 ^ n > 0` is an instance of the general fact that if `m` and `n` are any natural numbers with `m > 0`, then `m ^ n > 0`. Maybe that fact is in Lean's library:

```
example (m n : Nat) (h : m > 0) : m ^ n > 0 := by library_search
```

The `library_search` tactic comes up with `exact pow_pos h n`, and `#check @pow_pos` gives the result

```
@pow_pos : ∀ {α : Type u_1} [inst : StrictOrderedSemiring α] {a : α},
             0 < a → ∀ (n : ℕ), 0 < a ^ n
```

That means that we can use `pow_pos` to prove `2 ^ n > 0`, but first we'll need to prove that `2 > 0`. We now have all the pieces we need; putting them together leads to this proof:

```
theorem Example_6_3_1 : ∀ n ≥ 4, fact n > 2 ^ n := by
  by_induc
  · -- Base Case
    norm_num
    done
  · -- Induction Step
```

```
      fix n : Nat
      assume h1 : n ≥ 4
      assume ih : fact n > 2 ^ n
      have h2 : n + 1 > 0 := by linarith
      have h3 : n + 1 > 2 := by linarith
      have h4 : 2 > 0 := by linarith
      have h5 : 2 ^ n > 0 := pow_pos h4 n
      show fact (n + 1) > 2 ^ (n + 1) from
        calc fact (n + 1)
            = (n + 1) * fact n := by rfl
          _ > (n + 1) * 2 ^ n := Nat.mul_lt_mul_of_pos_left ih h2
          _ > 2 * 2 ^ n := Nat.mul_lt_mul_of_pos_right h3 h5
          _ = 2 ^ (n + 1) := by ring
      done
    done
```

The next example in *HTPI* is a proof of one of the laws of exponents: `a ^ (m + n) = a ^ m * a ^ n`. Lean's definition of exponentiation with natural number exponents is recursive. For some reason, the definitions are slightly different for different kinds of bases. The definitions Lean uses are essentially as follows:

```
-- For natural numbers x and n, x ^ n = nat_pow x n:
def nat_pow (x n : Nat) : Nat :=
  match n with
    | 0 => 1
    | k + 1 => (nat_pow x k) * x

-- For a real number x and a natural number n, x ^ n = real_pow x n:
def real_pow (x : Real) (n : Nat) : Real :=
  match n with
    | 0 => 1
    | k + 1 => x * (real_pow x k)
```

Let's prove the addition law for exponents:

```
theorem Example_6_3_2_cheating : ∀ (a : Real) (m n : Nat),
    a ^ (m + n) = a ^ m * a ^ n := by
  fix a : Real; fix m : Nat; fix n : Nat
  ring
  done
```

Well, that wasn't really fair. The `ring` tactic knows the laws of exponents, so it has no trouble proving this theorem. But we want to know *why* the law holds, so let's see if we can prove it

without using `ring`. The following proof is essentially the same as the proof in *HTPI*:

```
theorem Example_6_3_2 : ∀ (a : Real) (m n : Nat),
    a ^ (m + n) = a ^ m * a ^ n := by
  fix a : Real; fix m : Nat
    --Goal: ∀ (n : Nat), a ^ (m + n) = a ^ m * a ^ n
  by_induc
  · -- Base Case
    show a ^ (m + 0) = a ^ m * a ^ 0 from
      calc a ^ (m + 0)
          = a ^ m := by rfl
        _ = a ^ m * 1 := (mul_one (a ^ m)).symm
        _ = a ^ m * a ^ 0 := by rfl
    done
  · -- Induction Step
    fix n : Nat
    assume ih : a ^ (m + n) = a ^ m * a ^ n
    show a ^ (m + (n + 1)) = a ^ m * a ^ (n + 1) from
      calc a ^ (m + (n + 1))
          = a ^ ((m + n) + 1) := by rw [add_assoc]
        _ = a * a ^ (m + n) := by rfl
        _ = a * (a ^ m * a ^ n) := by rw [ih]
        _ = a ^ m * (a * a ^ n) :=
            by rw [←mul_assoc, mul_comm a, mul_assoc]
        _ = a ^ m * (a ^ (n + 1)) := by rfl
    done
  done
```

Finally, we'll prove the theorem in Example 6.3.4 of *HTPI*, which again involves exponentiation with natural number exponents. Here's the beginning of the proof:

Lean File

```
theorem Example_6_3_4 : ∀ (x : Real), x > -1 →
    ∀ (n : Nat), (1 + x) ^ n ≥ 1 + n * x := by
  fix x : Real
  assume h1 : x > -1
  done
```

Tactic State in Infoview

```
x : ℝ
h1 : x > -1
⊢ ∀ (n : ℕ),
    (1 + x) ^ n ≥
      1 + ↑n * x
```

Look carefully at the goal in the tactic state. Why is there a ↑ before the last n? The reason has to do with types. The variable x has type `Real` and n has type `Nat`, so how can Lean multiply n by x? Remember, in Lean, the natural numbers are not a subset of the real numbers. The two types are completely separate, but for each natural number, there is a corresponding real

number. To multiply n by x, Lean had to convert n to the corresponding real number, through a process called *coercion*. The notation ↑n denotes the result of *coercing* (or *casting*) n to another type—in this case, Real. Since ↑n and x are both real numbers, Lean can use the multiplication operation on the real numbers to multiply them. As we will see, the need for coercion in this example will make the proof a bit more complicated, because we'll need to use some theorems about coercions. Theorems about coercion of natural numbers to some other type often have names that start Nat.cast.

Continuing with the proof, since exponentiation is defined recursively, let's try mathematical induction:

Lean File

```
theorem Example_6_3_4 : ∀ (x : Real), x > -1 →
    ∀ (n : Nat), (1 + x) ^ n ≥ 1 + n * x := by
  fix x : Real
  assume h1 : x > -1
  by_induc
  · -- Base Case

    done
  · -- Induction Step

    done
  done
```

Tactic State in Infoview

```
case Base_Case
x : ℝ
h1 : x > -1
⊢ (1 + x) ^ 0 ≥
  1 + ↑0 * x
```

You might think that linarith could prove the goal for the base case, but it can't. The problem is the ↑0, which denotes the result of coercing the natural number 0 to a real number. Of course, that should be the real number 0, but is it? Yes, but the linarith tactic doesn't know that. The theorem Nat.cast_zero says that ↑0 = 0 (where the 0 on the right side of the equation is the *real number* 0), so the tactic rewrite [Nat.cast_zero] will convert ↑0 to 0. After that step, linarith can complete the proof of the base case, and we can start on the induction step.

Lean File

Tactic State in Infoview

```
theorem Example_6_3_4 : ∀ (x : Real), x > -1 →
    ∀ (n : Nat), (1 + x) ^ n ≥ 1 + n * x := by
  fix x : Real
  assume h1 : x > -1
  by_induc
  · -- Base Case
    rewrite [Nat.cast_zero]
    linarith
    done
  · -- Induction Step
    fix n : Nat
    assume ih : (1 + x) ^ n ≥ 1 + n * x
    done
  done
```

```
case Induction_Step
x : ℝ
h1 : x > -1
n : ℕ
ih : (1 + x) ^ n ≥
    1 + ↑n * x
⊢ (1 + x) ^ (n + 1) ≥
  1 + ↑(n + 1) * x
```

Once again, there's a complication caused by coercion. The inductive hypothesis talks about ↑n, but the goal involves ↑(n + 1). What is the relationship between these? Surely it should be the case that ↑(n + 1) = ↑n + 1; that is, the result of coercing the natural number n + 1 to a real number should be one larger than the result of coercing n to a real number. The theorem Nat.cast_succ says exactly that, so rewrite [Nat.cast_succ] will change the ↑(n + 1) in the goal to ↑n + 1. (The number n + 1 is sometimes called the *successor* of n, and succ is short for "successor.") With that change, we can continue with the proof. The following proof is modeled on the proof in *HTPI*.

```
theorem Example_6_3_4 : ∀ (x : Real), x > -1 →
    ∀ (n : Nat), (1 + x) ^ n ≥ 1 + n * x := by
  fix x : Real
  assume h1 : x > -1
  by_induc
  · -- Base Case
    rewrite [Nat.cast_zero]
    linarith
    done
  · -- Induction Step
    fix n : Nat
    assume ih : (1 + x) ^ n ≥ 1 + n * x
    rewrite [Nat.cast_succ]
    show (1 + x) ^ (n + 1) ≥ 1 + (↑n + 1) * x from
      calc (1 + x) ^ (n + 1)
          = (1 + x) * (1 + x) ^ n := by rfl
```

```
        _ ≥ (1 + x) * (1 + n * x) := sorry
        _ = 1 + x + n * x + n * x * x := by ring
        _ ≥ 1 + x + n * x + 0 := sorry
        _ = 1 + (↑n + 1) * x := by ring
    done
  done
```

Note that in the calculational proof, each n that is multiplied by x is really ↑n, but we don't need to say so explicitly; Lean fills in coercions automatically when they are required.

All that's left is to replace the two occurrences of sorry with justifications. We'll use two theorems:

```
@mul_le_mul_of_nonneg_left : ∀ {α : Type u_1} {a b c : α} [inst : Mul α]
        [inst_1 : Zero α] [inst_2 : Preorder α] [inst_3 : PosMulMono α],
        b ≤ c → 0 ≤ a → a * b ≤ a * c

@add_le_add_left : ∀ {α : Type u_1} [inst : Add α] [inst_1 : LE α]
        [inst_2 : CovariantClass α α (fun (x x_1 : α) => x + x_1)
        fun (x x_1 : α) => x ≤ x_1] {b c : α},
        b ≤ c → ∀ (a : α), a + b ≤ a + c
```

As usual, we don't need to pay much attention to the long lists of implicit arguments; what is important is the last line of each statement. Those lines tell us that mul_le_mul_of_nonneg_left can be applied to the inductive hypothesis to justify the first sorry step, if we can prove 1 + x ≥ 0. And then add_le_add_left will justify the second sorry step, if we can prove n * x * x ≥ 0. So here is our next attempt:

```
  theorem Example_6_3_4 : ∀ (x : Real), x > -1 →
      ∀ (n : Nat), (1 + x) ^ n ≥ 1 + n * x := by
    fix x : Real
    assume h1 : x > -1
    by_induc
    · -- Base Case
      rewrite [Nat.cast_zero]
      linarith
      done
    · -- Induction Step
      fix n : Nat
      assume ih : (1 + x) ^ n ≥ 1 + n * x
      rewrite [Nat.cast_succ]
      have h2 : 1 + x ≥ 0 := by linarith
      have h3 : n * x * x ≥ 0 := sorry
```

```
        show (1 + x) ^ (n + 1) ≥ 1 + (↑n + 1) * x from
          calc (1 + x) ^ (n + 1)
              = (1 + x) * (1 + x) ^ n := by rfl
            _ ≥ (1 + x) * (1 + n * x) := mul_le_mul_of_nonneg_left ih h2
            _ = 1 + x + n * x + n * x * x := by ring
            _ ≥ 1 + x + n * x + 0 := add_le_add_left h3 _
            _ = 1 + (↑n + 1) * x := by ring
        done
      done
```

The `linarith` tactic is able to prove `1 + x ≥ 0`, since it follows easily from `h1 : x > -1`. But how can we prove `n * x * x ≥ 0`? We start with the fact that the square of any real number is nonnegative:

```
  @mul_self_nonneg : ∀ {α : Type u_1} [inst : LinearOrderedRing α]
                       (a : α), 0 ≤ a * a
```

So `mul_self_nonneg x` is a proof of `x * x ≥ 0`. To get `n * x * x ≥ 0`, which means `↑n * x * x ≥ 0`, we'll need another application of `mul_le_mul_of_nonneg_left`, and we'll need to know `↑n ≥ 0`. But if we use the `have` tactic to assert `↑n ≥ 0`, then Lean won't know what we mean. The problem is that this last statement is ambiguous. The notation `↑n` means "n coerced to some other type," but what is the other type? Lean can usually figure this out from context, but in the statement `↑n ≥ 0` there is no context to tell Lean what type to use. So we must say `(↑n : Real) ≥ 0` to eliminate the ambiguity. The theorem `Nat.cast_nonneg` proves this statement, providing the last piece of the puzzle. Here's the complete proof:

```
  theorem Example_6_3_4 : ∀ (x : Real), x > -1 →
      ∀ (n : Nat), (1 + x) ^ n ≥ 1 + n * x := by
    fix x : Real
    assume h1 : x > -1
    by_induc
    · -- Base Case
      rewrite [Nat.cast_zero]
      linarith
      done
    · -- Induction Step
      fix n : Nat
      assume ih : (1 + x) ^ n ≥ 1 + n * x
      rewrite [Nat.cast_succ]
      have h2 : 1 + x ≥ 0 := by linarith
      have h3 : n * x * x ≥ 0 := by
        have h4 : x * x ≥ 0 := mul_self_nonneg x
```

```
      have h5 : (↑n : Real) ≥ 0 := Nat.cast_nonneg n
      show n * x * x ≥ 0 from
        calc n * x * x
            = n * (x * x) := mul_assoc _ _ _
          _ ≥ n * 0 := mul_le_mul_of_nonneg_left h4 h5
          _ = 0 := by ring
      done
    show (1 + x) ^ (n + 1) ≥ 1 + (↑n + 1) * x from
      calc (1 + x) ^ (n + 1)
          = (1 + x) * (1 + x) ^ n := by rfl
        _ ≥ (1 + x) * (1 + n * x) := mul_le_mul_of_nonneg_left ih h2
        _ = 1 + x + n * x + n * x * x := by ring
        _ ≥ 1 + x + n * x + 0 := add_le_add_left h3 _
        _ = 1 + (↑n + 1) * x := by ring
    done
  done
```

Before ending this section, we'll return to a topic left unexplained before. We can now describe how `Sum i from k to n, f i` is defined. The key is a function `sum_less`, which is defined by recursion:

```
def sum_less {A : Type} [AddZeroClass A] (x : Nat) (f : Nat → A) : A :=
  match x with
    | 0 => 0
    | n + 1 => sum_less n f + f n
```

To get an idea of what this definition means, let's try evaluating `sum_less 3 f`:

```
sum_less 3 f = sum_less 2 f + f 2
             = sum_less 1 f + f 1 + f 2
             = sum_less 0 f + f 0 + f 1 + f 2
             = 0 + f 0 + f 1 + f 2
```

So `sum_less 3 f` adds up all values of `f i`, for `i < 3`. More generally, `sum_less x f` gives the sum of all values `f i` for `i < x`. (The implicit arguments say that the type of `f i` can be any type for which `0` and `+` make sense.) The notation `Sum i from k to n, f i` is now defined to be a shorthand for `sum_less (n + 1 - k) (fun x => f (k + x))`. We'll leave it to you to puzzle out why that gives the desired result.

Exercises

1.
```
theorem Exercise_6_3_4 : ∀ (n : Nat),
    3 * (Sum i from 0 to n, (2 * i + 1) ^ 2) =
    (n + 1) * (2 * n + 1) * (2 * n + 3) := sorry
```

2.
```
theorem Exercise_6_3_7b (f : Nat → Real) (c : Real) : ∀ (n : Nat),
    Sum i from 0 to n, c * f i = c * Sum i from 0 to n, f i := sorry
```

3.
```
theorem fact_pos : ∀ (n : Nat), fact n ≥ 1 := sorry
```

4.
```
--Hint:  Use the theorem fact_pos from the previous exercise
theorem Exercise_6_3_13a (k : Nat) : ∀ (n : Nat),
    fact (k ^ 2 + n) ≥ k ^ (2 * n) := sorry
```

5.
```
--Hint:  Use the theorem in the previous exercise.
--You may find it useful to first prove a lemma:
--∀ (k : Nat), 2 * k ^ 2 + 1 ≥ k
theorem Exercise_6_3_13b (k : Nat) : ∀ n ≥ 2 * k ^ 2,
    fact n ≥ k ^ n := sorry
```

6. A sequence is defined recursively as follows:

```
def seq_6_3_15 (x : Nat) : Int :=
  match x with
    | 0 => 0
    | n + 1 => 2 * seq_6_3_15 n + n
```

Prove the following theorem about this sequence:

```
theorem Exercise_6_3_15 : ∀ (n : Nat),
    seq_6_3_15 n = 2 ^ n - n - 1 := sorry
```

7. A sequence is defined recursively as follows:

```
def seq_6_3_16 (x : Nat) : Nat :=
  match x with
    | 0 => 2
    | n + 1 => (seq_6_3_16 n) ^ 2
```

Find a formula for `seq_6_3_16 n`. Fill in the blank in the theorem below with your formula and then prove the theorem.

```
theorem Exercise_6_3_16 : ∀ (n : Nat),
    seq_6_3_16 n = ___ := sorry
```

6.4. Strong Induction

In the induction step of a proof by mathematical induction, we prove that a natural number has some property from the assumption that the previous number has the property. Section 6.4 of *HTPI* introduces a version of mathematical induction in which we get to assume that *all* smaller numbers have the property. Since this is a stronger assumption, this version of induction is called *strong induction*. Here is how strong induction works:

To prove a goal of the form ∀ (n : Nat), P n:

  Prove that ∀ (n : Nat), (∀ k < n, P k) → P n.

To write a proof by strong induction in Lean, we use the tactic `by_strong_induc`, whose effect on the tactic state can be illustrated as follows.

Tactic State Before Using Strategy	Tactic State After Using Strategy

```
  ⋮
⊢ ∀ (n : Nat), P n
```

```
  ⋮
⊢  ∀ (n : Nat),
    (∀ (n_1 : Nat), n_1 < n → P n_1)
      → P n
```

To illustrate this, we begin with Example 6.4.1 of *HTPI*.

```
theorem Example_6_4_1 : ∀ m > 0, ∀ (n : Nat),
    ∃ (q r : Nat), n = q * m + r ∧ r < m
```

Imitating the strategy of the proof in *HTPI*, we let `m` be an arbitrary natural number, assume `m > 0`, and then prove the statement `∀ (n : Nat), ∃ (q r : Nat), n = q * m + r ∧ r < m` by strong induction. That means that after introducing an arbitrary natural number n, we assume the inductive hypothesis, which says that `∀ k < n, ∃ (q r : Nat), k = q * m + r ∧ r < m`.

```
theorem Example_6_4_1 : ∀ m > 0, ∀ (n : Nat),
    ∃ (q r : Nat), n = q * m + r ∧ r < m := by
```

```
    fix m : Nat
    assume h1 : m > 0
    by_strong_induc
    fix n : Nat
    assume ih : ∀ k < n, ∃ (q r : Nat), k = q * m + r ∧ r < m
    done
```

Our goal now is to prove that `∃ (q r : Nat), n = q * m + r ∧ r < m`. Although strong induction does not require a base case, it is not uncommon for proofs by strong induction to involve reasoning by cases. The proof in *HTPI* uses the cases `n < m` and `m ≤ n`. If `n < m`, then the proof is easy: the numbers `q = 0` and `r = n` clearly have the required properties. In the case `m ≤ n`, the proof in *HTPI* proceeds by applying the inductive hypothesis to the number `k = n - m`. Since `m > 0`, we have `k < n`, so the inductive hypothesis can be applied. Notice that if `m > 1`, then `n - m` is not the number immediately preceding `n`; that's why this proof uses strong induction rather than ordinary induction.

Since `n` and `m` have type `Nat`, the subtraction in the formula `n - m` is natural number subtraction, which, as we've seen, can be difficult to work with. But in this proof we use this formula only in the case when `m ≤ n`, and in that case the subtraction is unproblematic. Indeed, there are theorems in Lean that tell us that if `m ≤ n`, then `n - m` has the properties we expect. We will use two such theorems:

```
@Nat.sub_lt_of_pos_le : ∀ (a b : ℕ), 0 < a → a ≤ b → b - a < b

@Nat.sub_add_cancel : ∀ {n m : ℕ}, m ≤ n → n - m + m = n
```

With that preparation, we are ready to complete the proof.

```
  theorem Example_6_4_1 : ∀ m > 0, ∀ (n : Nat),
      ∃ (q r : Nat), n = q * m + r ∧ r < m := by
    fix m : Nat
    assume h1 : m > 0
    by_strong_induc
    fix n : Nat
    assume ih : ∀ k < n, ∃ (q r : Nat), k = q * m + r ∧ r < m
    by_cases h2 : n < m
    · -- Case 1. h2: n < m
      apply Exists.intro 0
      apply Exists.intro n      --Goal: n = 0 * m + n ∧ n < m
      apply And.intro _ h2
      ring
      done
    · -- Case 2. h2: ¬n < m
```

```
      have h3 : m ≤ n := by linarith
      let k : Nat := n - m
      have h4 : k < n := Nat.sub_lt_of_pos_le m n h1 h3
      have h5 : ∃ (q r : Nat), k = q * m + r ∧ r < m := ih k h4
      obtain (q' : Nat)
        (h6 : ∃ (r : Nat), k = q' * m + r ∧ r < m) from h5
      obtain (r' : Nat) (h7 : k = q' * m + r' ∧ r' < m) from h6
      apply Exists.intro (q' + 1)
      apply Exists.intro r'      --Goal: n = (q' + 1) * m + r' ∧ r' < m
      apply And.intro _ h7.right
      show n = (q' + 1) * m + r' from
        calc n
            = k + m := (Nat.sub_add_cancel h3).symm
          _ = q' * m + r' + m := by rw [h7.left]
          _ = (q' + 1) * m + r' := by ring
      done
    done
```

The numbers q and r in `Example_6_4_1` are called the *quotient* and *remainder* when n is divided by m. Lean knows how to compute these numbers: if n and m are natural numbers, then in Lean, n / m denotes the quotient when n is divided by m, and n % m denotes the remainder. (The number n % m is also sometimes called n *modulo* m.) And Lean knows theorems stating that these numbers have the properties specified in `Example_6_4_1`:

```
@Nat.div_add_mod : ∀ (m n : ℕ), n * (m / n) + m % n = m
```

```
@Nat.mod_lt : ∀ (x : ℕ) {y : ℕ}, y > 0 → x % y < y
```

By the way, although we are unlikely to want to use the notation n / 0 or n % 0, Lean uses the definitions n / 0 = 0 and n % 0 = n. As a result, the equation n * (m / n) + m % n = m is true even if n = 0. That's why the theorem `Nat.div_add_mod` doesn't include a requirement that n > 0.

There is also a strong form of recursion. As an example of this, here is a recursive definition of a sequence of numbers called the *Fibonacci numbers*:

```
def Fib (x : Nat) : Nat :=
  match x with
    | 0 => 0
    | 1 => 1
    | n + 2 => Fib n + Fib (n + 1)
```

Notice that the formula for Fib (n + 2) involves the *two* previous values of Fib, not just the immediately preceding value. That is the sense in which the recursion is *strong*. Not

surprisingly, theorems about the Fibonacci numbers are often proven by induction—either ordinary or strong. We'll illustrate this with a proof that `∀ (n : Nat), Fib n < 2 ^ n`. This time we'll need to treat the cases `n = 0` and `n = 1` separately, since these values are treated separately in the definition of `Fib n`. We'll also use the theorems `add_lt_add_left` and `add_lt_add_right`; these are similar to the theorem `add_le_add_left`, which we used in `Example_6_3_4`.

```
example : ∀ (n : Nat), Fib n < 2 ^ n := by
  by_strong_induc
  fix n : Nat
  assume ih : ∀ (k : Nat), k < n → Fib k < 2 ^ k
  by_cases h1 : n = 0
  · -- Case 1. h1: n = 0
    rewrite [h1]            --Goal: Fib 0 < 2 ^ 0
    norm_num
    done
  · -- Case 2. h1: n ≠ 0
    by_cases h2 : n = 1
    · -- Case 2.1. h2: n = 1
      rewrite [h2]          --Goal: Fib 1 < 2 ^ 1
      norm_num
      done
    · -- Case 2.2. h2: n ≠ 1
      have h3 : 2 ≤ n := sorry
      let k : Nat := n - 2
      have h4 : k + 2 = n := Nat.sub_add_cancel h3
      have h5 : k < n := by linarith
      have h6 : Fib k < 2 ^ k := ih k h5
      have h7 : k + 1 < n := by linarith
      have h8 : Fib (k + 1) < 2 ^ (k + 1) := ih (k + 1) h7
      show Fib n < 2 ^ n from
        calc Fib n
            = Fib (k + 2) := by rw [h4]
          _ = Fib k + Fib (k + 1) := by rfl
          _ < 2 ^ k + Fib (k + 1) := add_lt_add_right h6 _
          _ < 2 ^ k + 2 ^ (k + 1) := add_lt_add_left h8 _
          _ = 2 ^ k * 3 := by ring
          _ ≤ 2 ^ k * 4 := sorry
          _ = 2 ^ (k + 2) := by ring
          _ = 2 ^ n := by rw [h4]
      done
  done
```

```
    done
```

There are two steps that require further justification. The statement `2 ≤ n` should follow from
`h1 : n ≠ 0` and `h2 : n ≠ 1`. To prove it, we'll use two theorems:

```
@Nat.pos_of_ne_zero : ∀ {n : ℕ}, n ≠ 0 → 0 < n

@lt_of_le_of_ne' : ∀ {α : Type u_1} [inst : PartialOrder α] {a b : α},
                   a ≤ b → b ≠ a → a < b
```

Using the first of these theorems, `Nat.pos_of_ne_zero h1` is a proof of `0 < n`. But for natural
numbers `a` and `b`, Lean treats `a < b` as meaning the same thing as `a + 1 ≤ b`, so this is also a
proof of `1 ≤ n`. Applying the theorem `lt_of_le_of_ne'` to this conclusion and `h2` proves `1 < n`,
which is definitionally equal to `2 ≤ n`.

To eliminate the second `sorry` we'll use the theorem:

```
@Nat.mul_le_mul_left : ∀ {n m : ℕ} (k : ℕ), n ≤ m → k * n ≤ k * m
```

To apply the theorem, we'll need to know `3 ≤ 4`, which can be proven by `linarith`. Here's the
complete proof:

```
example : ∀ (n : Nat), Fib n < 2 ^ n := by
  by_strong_induc
  fix n : Nat
  assume ih : ∀ (k : Nat), k < n → Fib k < 2 ^ k
  by_cases h1 : n = 0
  · -- Case 1. h1: n = 0
    rewrite [h1]          --Goal: Fib 0 < 2 ^ 0
    norm_num
    done
  · -- Case 2. h1: n ≠ 0
    by_cases h2 : n = 1
    · -- Case 2.1. h2: n = 1
      rewrite [h2]        --Goal: Fib 1 < 2 ^ 1
      norm_num
      done
    · -- Case 2.2. h2: n ≠ 1
      have h3 : 2 ≤ n := by
        have h4 : 1 ≤ n := Nat.pos_of_ne_zero h1
        show 2 ≤ n from lt_of_le_of_ne' h4 h2
        done
      let k : Nat := n - 2
      have h4 : k + 2 = n := Nat.sub_add_cancel h3
```

```
      have h5 : k < n := by linarith
      have h6 : Fib k < 2 ^ k := ih k h5
      have h7 : k + 1 < n := by linarith
      have h8 : Fib (k + 1) < 2 ^ (k + 1) := ih (k + 1) h7
      have h9 : 3 ≤ 4 := by linarith
      show Fib n < 2 ^ n from
        calc Fib n
            = Fib (k + 2) := by rw [h4]
          _ = Fib k + Fib (k + 1) := by rfl
          _ < 2 ^ k + Fib (k + 1) := add_lt_add_right h6 _
          _ < 2 ^ k + 2 ^ (k + 1) := add_lt_add_left h8 _
          _ = 2 ^ k * 3 := by ring
          _ ≤ 2 ^ k * 4 := Nat.mul_le_mul_left _ h9
          _ = 2 ^ (k + 2) := by ring
          _ = 2 ^ n := by rw [h4]
      done
    done
  done
```

As with ordinary induction, strong induction can be useful for proving statements that do not at first seem to have the form `∀ (n : Nat), ...`. To illustrate this, we'll prove the *well-ordering principle*, which says that if a set `S : Set Nat` is nonempty, then it has a smallest element. We'll prove the contrapositive: if `S` has no smallest element, then it is empty. To say that `S` is empty means `∀ (n : Nat), n ∉ S`, and that's the statement to which we will apply strong induction.

```
theorem well_ord_princ (S : Set Nat) (h1 : ∃ (x : Nat), x ∈ S) :
    ∃ (x : Nat), x ∈ S ∧ ∀ (y : Nat), y ∈ S → x ≤ y := by
  contradict h1 with h2
    --h2: ¬∃ (x : Nat), x ∈ S ∧ ∀ (y : Nat), y ∈ S → x ≤ y
  quant_neg                       --Goal: ∀ (x : Nat), ¬x ∈ S
  by_strong_induc
  fix n : Nat
  assume ih : ∀ k < n, k ∉ S  --Goal: ¬n ∈ S
  contradict h2 with h3         --h3: n ∈ S
    --Goal: ∃ (x : Nat), x ∈ S ∧ ∀ (y : Nat), y ∈ S → x ≤ y
  apply Exists.intro n
  apply And.intro h3            --Goal: ∀ (y : Nat), y ∈ S → n ≤ y
  fix y : Nat
  assume h4 : y ∈ S
  have h5 : y < n → ¬y ∈ S := ih y
  contrapos at h5               --h5: y ∈ S → ¬y < n
```

```
    have h6 : ¬y < n := h5 h4
    linarith
    done
```

Section 6.4 of *HTPI* ends with an example of an application of the well ordering principle. The example gives a proof that $\sqrt{2}$ is irrational. If $\sqrt{2}$ were rational, then there would be natural numbers p and q such that $q \neq 0$ and $p/q = \sqrt{2}$, and therefore $p^2 = 2q^2$. So we can prove that $\sqrt{2}$ is irrational by showing that there do not exist natural numbers p and q such that $q \neq 0$ and $p^2 = 2q^2$.

The proof uses some definitions and theorems from the exercises of Section 6.1:

```
def nat_even (n : Nat) : Prop := ∃ (k : Nat), n = 2 * k

def nat_odd (n : Nat) : Prop := ∃ (k : Nat), n = 2 * k + 1

theorem Exercise_6_1_16a1 :
    ∀ (n : Nat), nat_even n ∨ nat_odd n

theorem Exercise_6_1_16a2 :
    ∀ (n : Nat), ¬(nat_even n ∧ nat_odd n)
```

We will also use a lemma, whose proof we leave as an exercise for you:

```
--Hint: Use Exercise_6_1_16a1 and Exercise_6_1_16a2
lemma sq_even_iff_even (n : Nat) : nat_even (n * n) ↔ nat_even n := sorry
```

And we'll need another theorem that we haven't seen before:

```
@mul_left_cancel_iff_of_pos : ∀ {α : Type u_1} {a b c : α}
                    [inst : MulZeroClass α] [inst_1 : PartialOrder α]
                    [inst_2 : PosMulMonoRev α],
                    0 < a → (a * b = a * c ↔ b = c)
```

To show that $\sqrt{2}$ is irrational, we will prove the statement

```
    ¬∃ (q p : Nat), p * p = 2 * (q * q) ∧ q ≠ 0
```

We proceed by contradiction. If this statement were false, then the set

```
    S = { q : Nat | ∃ (p : Nat), p * p = 2 * (q * q) ∧ q ≠ 0 }
```

would be nonempty, and therefore, by the well ordering principle, it would have a smallest element. We then show that this leads to a contradiction. Here is the proof.

```
theorem sqrt_2_irrat :
    ¬∃ (q p : Nat), p * p = 2 * (q * q) ∧ q ≠ 0 := by
  let S : Set Nat :=
    { q : Nat | ∃ (p : Nat), p * p = 2 * (q * q) ∧ q ≠ 0 }
  by_contra h1
  have h2 : ∃ (q : Nat), q ∈ S := h1
  have h3 : ∃ (x : ℕ), x ∈ S ∧ ∀ (y : ℕ), y ∈ S → x ≤ y :=
    well_ord_princ S h2
  obtain (q : Nat) (h4 : q ∈ S ∧ ∀ (y : ℕ), y ∈ S → q ≤ y) from h3
  have h4l : q ∈ S := h4.left
  define at h4l      --h4l: ∃ (p : ℕ), p * p = 2 * (q * q) ∧ q ≠ 0
  obtain (p : Nat) (h5 : p * p = 2 * (q * q) ∧ q ≠ 0) from h4l
  have h5l : p * p = 2 * (q * q) := h5.left
  have h6 : nat_even (p * p) := Exists.intro (q * q) h5l
  rewrite [sq_even_iff_even p] at h6     --h6: nat_even p
  obtain (p' : Nat) (h7 : p = 2 * p') from h6
  have h8 : 2 * (2 * (p' * p')) = 2 * (q * q) := by
    rewrite [←h5l, h7]
    ring
    done
  have h9 : 2 > 0 := by norm_num
  rewrite [mul_left_cancel_iff_of_pos h9] at h8
    --h8: 2 * (p' * p') = q * q
  have h10 : nat_even (q * q) := Exists.intro (p' * p') h8.symm
  rewrite [sq_even_iff_even q] at h10    --h10: nat_even q
  obtain (q' : Nat) (h11 : q = 2 * q') from h10
  have h12 : 2 * (p' * p') = 2 * (2 * (q' * q')) := by
    rewrite [h8, h11]
    ring
    done
  rewrite [mul_left_cancel_iff_of_pos h9] at h12
    --h12: p' * p' = 2 * (q' * q')
  have h13 : q' ≠ 0 := by
    contradict h5.right with h14
    rewrite [h11, h14]
    rfl
    done
  have h14 : q' ∈ S := Exists.intro p' (And.intro h12 h13)
  have h15 : q ≤ q' := h4.right q' h14
  rewrite [h11] at h15        --h15: 2 * q' ≤ q'
  contradict h13
  linarith
```

```
    done
```

Exercises

1.
```
--Hint: Use Exercise_6_1_16a1 and Exercise_6_1_16a2
lemma sq_even_iff_even (n : Nat) :
    nat_even (n * n) ↔ nat_even n := sorry
```

2.
```
--This theorem proves that the square root of 6 is irrational
theorem Exercise_6_4_4a :
    ¬∃ (q p : Nat), p * p = 6 * (q * q) ∧ q ≠ 0 := sorry
```

3.
```
theorem Exercise_6_4_5 :
    ∀ n ≥ 12, ∃ (a b : Nat), 3 * a + 7 * b = n := sorry
```

4.
```
theorem Exercise_6_4_7a : ∀ (n : Nat),
    (Sum i from 0 to n, Fib i) + 1 = Fib (n + 2) := sorry
```

5.
```
theorem Exercise_6_4_7c : ∀ (n : Nat),
    Sum i from 0 to n, Fib (2 * i + 1) = Fib (2 * n + 2) := sorry
```

6.
```
theorem Exercise_6_4_8a : ∀ (m n : Nat) ,
    Fib (m + n + 1) = Fib m * Fib n + Fib (m + 1) * Fib (n + 1) := sorry
```

7.
```
theorem Exercise_6_4_8d : ∀ (m k : Nat), Fib m | Fib (m * k) := sorry
```

Hint for #7: Let `m` be an arbitrary natural number, and then use induction on `k`. For the induction step, you must prove `Fib m | Fib (m * (k + 1))`. If `m = 0 ∨ k = 0`, then this is easy. If not, then show that you can rewrite `m * (k + 1)` as `(m * k - 1) + m + 1`, and then apply Exercise_6_4_8a.

8.
```
def Fib_like (x : Nat) : Nat :=
  match x with
    | 0 => 1
    | 1 => 2
    | n + 2 => 2 * (Fib_like n) + Fib_like (n + 1)

theorem Fib_like_formula : ∀ (n : Nat), Fib_like n = 2 ^ n := sorry
```

9.
```
def triple_rec (x : Nat) : Nat :=
  match x with
    | 0 => 0
    | 1 => 2
    | 2 => 4
    | n + 3 => 4 * triple_rec n +
                 6 * triple_rec (n + 1) + triple_rec (n + 2)

theorem triple_rec_formula :
    ∀ (n : Nat), triple_rec n = 2 ^ n * Fib n := sorry
```

10. In this exercise you will prove that the numbers q and r in Example_6_4_1 are unique. It is helpful to prove a lemma first.

```
lemma quot_rem_unique_lemma {m q r q' r' : Nat}
    (h1 : q * m + r = q' * m + r') (h2 : r' < m) : q ≤ q' := sorry

theorem quot_rem_unique (m q r q' r' : Nat)
    (h1 : q * m + r = q' * m + r') (h2 : r < m) (h3 : r' < m) :
    q = q' ∧ r = r' := sorry
```