Anne Baanen
Alexander Bentkamp
Jasmin Blanchette
Johannes Hölzl
Jannis Limperg

The Hitchhiker's Guide to Logical Verification

2023 Standard Edition
(November 21, 2023)

lean-forward.github.io/
hitchhikers-guide/2023

Contents

Preface

Formal proof assistants are software tools designed to help their users carry out computer-checked proofs. We usually call them *proof assistants*, or *interactive theorem provers*, but a frustrated student coined the phrase "proof-preventing beasts," and dictation software occasionally misunderstands "theorem prover" as "fear improver." Consider yourself warned.

Rigorous and Formal Proofs Interactive theorem proving has its own terminology, starting with the notion of "proof." A *formal proof* is a logical argument expressed in a logical formalism. In this context, "formal" means "logical" or "logic-based." Logicians—the mathematicians of logics—carried out formal proofs on paper decades before the advent of computers, but nowadays formal proofs are almost always carried out using a proof assistant.

In contrast, an *informal proof* is what a mathematician would normally call a proof. These are often carried out in LaTeX or on a blackboard, and are also called "pen-and-paper proofs." The level of detail can vary a lot, and phrases such as "it is obvious that," "clearly," and "without loss of generality" move some of the proof burden onto the reader. A *rigorous proof* is a very detailed informal proof.

The main strength of proof assistants is that they help develop highly trustworthy, unambiguous proofs of mathematical statements, using a precise logic. They can be used to prove arbitrarily advanced results, going well beyond toy examples and logic puzzles. Formal proofs also help students understand what constitutes a valid definition or a valid proof. To quote Scott Aaronson:[1]

> I still remember having to grade hundreds of exams where the students started out by assuming what had to be proved, or filled page after page with gibberish in the hope that, somewhere in the mess, they *might* accidentally have said something correct.

When we develop a new theory, formal proofs can help us explore it. They are useful when we generalize, refactor, or otherwise modify an existing proof, in much the same way as a compiler helps us develop correct programs. They offer a high level of trustworthiness, making it easier for others to review them. In addition, formal proofs can form the basis of verified computational tools (e.g., verified computer algebra systems).

Success Stories There have been a number of proof assistant success stories in mathematics and computer science. Some landmark results in the formalization of mathematics have been the proof of the four-color theorem by Gonthier et

[1] `https://www.scottaaronson.com/teaching.pdf`

al. [6], the proof of the odd-order theorem by Gonthier et al. [7], the proof of the Kepler conjecture by Hales et al. [10], and the definition of perfectoid spaces by Buzzard et al. [4]. The earliest work in this area was carried out by Nicolaas de Bruijn and his colleagues starting in the 1960s in a system called AUTOMATH.[2]

Today, few mathematicians use proof assistants, but this is slowly changing. For example, 29 participants of the Lean Together 2019 meeting in Amsterdam,[3] about formalization of mathematics, self-identified as mathematicians.

Most users of proof assistants are computer scientists. A few companies, including AMD [28] and Intel [11], have been using proof assistants to verify their designs. In academia, some of the milestones are the verifications of the operating system kernels seL4 [14] and CertiKOS [9] and the development of the verified compilers CompCert [17], JinjaThreads [20], and CakeML [16].

Proof Assistants There are dozens of proof assistants in development or use across the world. A list of the main ones follows, grouped by their logical foundation:

- set theory: Isabelle/ZF, Metamath, Mizar;
- simple type theory: HOL4, HOL Light, Isabelle/HOL;
- dependent type theory: Agda, Coq, Lean, Matita, PVS;
- Lisp-like first-order logic: ACL2.

For a history of proof assistants and interactive theorem proving, we refer to Harrison, Urban, and Wiedijk's highly informative chapter [12].

Lean Lean is a proof assistant developed primarily by Leonardo de Moura (Microsoft Research) since 2012. Its mathematical library, `mathlib`, was originally developed under the leadership of Jeremy Avigad (Carnegie Mellon University) but is now maintained and further extended by the user community [21].[4]

This guide uses Lean version 4.0.0-nightly-2023-08-19, `mathlib` revision 3689-23da0362e039, and a few extensions collected in a small library called `LoVelib`.[5] Although it is a research project, with some rough edges, there are several reasons why Lean is suitable to teach interactive theorem proving:

- It has a highly expressive, and very interesting, logic based on the calculus of inductive constructions, a dependent type theory.
- It is extended with classical axioms and quotient types, making it useful to verify mathematics.
- It includes a convenient metaprogramming framework, which can be used to program custom proof automation.
- It includes a modern user interface via a Visual Studio Code plugin.
- It has highly readable, fairly complete documentation.
- It is open source.

[2] `https://www.win.tue.nl/automath/`

[3] `https://lean-forward.github.io/lean-together/2019/index.html`

[4] `https://github.com/leanprover-community/mathlib4`

[5] `https://github.com/blanchette/logical_verification_2023/raw/main/lean/LoVelib.lean`

Lean's core library includes only basic algebraic definitions. More definitions are found in `mathlib`. Despite its name, `mathlib` is more than a mathematical library; it provides a lot of basic automation on top of Lean's core library, as one would expect from a modern proof assistant.

This Guide This guide was originally designed as a companion to the MSc-level course Logical Verification (LoVe) taught at the Vrije Universiteit Amsterdam. Its primary aim is to teach interactive theorem proving. Lean is the means, not an end in itself. As such, this guide is not designed to be a comprehensive Lean tutorial—for this, we recommend *Theorem Proving in Lean 4* [13]. This guide is also no substitute for doing the exercises and homework. Theorem proving is not for spectators; it can only be learned by doing.

Specifically, the aim is that you

- learn the fundamental theory and techniques of interactive theorem proving;
- learn how to use logic as a precise language for modeling systems and stating properties about them;
- become familiar with some areas in which proof assistants are successfully applied, such as functional programming, the semantics of imperative programming languages, and mathematics;
- develop practical skills that can be applyed in larger projects (whether personal or for an MSc or PhD or in industry);
- feel able to move to another proof assistant and apply what you have learned;
- get to understand the domain well enough to start reading relevant scientific papers published at international conferences such as Certified Programs and Proofs (CPP) and Interactive Theorem Proving (ITP) or in journals such as the *Journal of Automated Reasoning* (JAR).

Equipped with a good knowledge of Lean, it should be easy to move to another proof assistant based on dependent type theory, such as Agda or Coq, or to a system based on simple type theory, such as HOL4 or Isabelle/HOL.

An important characteristic of this guide, which it shares with Knuth's *TEXbook* [15], is that it does not always tell the truth. To simplify the exposition, some basic but false claims are made about Lean. Most of these statements are then rectified in later chapters. Like Knuth, we feel that "this technique of deliberate lying will actually make it easier for you to learn the ideas. Once you understand a simple but false rule, it will not be hard to supplement that rule with its exceptions."

The Lean files accompanying this guide can be found in a public repository.[6] The files' naming scheme follows this guide's chapters; thus, `LoVe07_EffectfulProgramming_Demo.lean` is the main file associated with Chapter 7 ("Effectful Programming"), reviewed in class, `LoVe07_EffectfulProgramming_ExerciseSheet.lean` is the exercise sheet, and `LoVe07_EffectfulProgramming_HomeworkSheet.lean` is the homework sheet.

We owe a huge debt to the authors of *Theorem Proving in Lean 4* [13] and *Concrete Semantics: With Isabelle/HOL* [23], who have taught us Lean and programming language semantics. Many of their ideas appear in this guide.

[6] `https://github.com/blanchette/logical_verification_2023`

We thank Robert Lewis and Assia Mahboubi for their useful comments on this guide's organization and focus. We thank Kiran Gopinathan and Ilya Sergey for sharing the anecdote reported in footnote 3 of Chapter 2 and letting us share it further. We thank Daniel Fabian for designing the first tablet-optimized version of this guide. We thank Paul Chisholm for authoring some of the comments in the associated Lean files. We thank Mark Summerfield for his many textual suggestions. Finally, we thank Chris Bailey, Kevin Buzzard, Paul Chisholm, Dominique Danco, Raufs Dunamalijevs, Wan Fokkink, Lina Gerlach, Robert Lewis, Pietro Monticone, Antonius Danny Reyes, Robert Schütz, Patrick Thomas, Huub Vromen, Floris Westerman, and Wijnand van Woerkom for reporting typos and some more serious errors they found in earlier editions of this guide. Should *you* find some mistakes in this text, please let us know.[7]

Special Symbols In this guide, we assume that you will be using Visual Studio Code and its Lean extension to edit `.lean` files. Visual Studio Code lets you enter Unicode symbols by entering backslash \ followed by an ASCII identifier. For example, →, ∀, or ∈ can be entered by typing `\->`, `\fo`, or `\in` and pressing the tab key or the space bar. We will freely use these notations. For reference, we provide a list of the main non-ASCII symbols that are used in this guide and, for each, one of its ASCII representations. By hovering over a symbol in Visual Studio Code while holding the control or command key pressed, you will see the different ways in which it can be entered.

¬	`\not`	∧	`\and`	∨	`\or`
→	`\->`	↔	`\<->`	∀	`\fo`
∃	`\ex`	≤	`\<=`	≥	`\>=`
≠	`\neq`	≈	`\~~`	×	`\x`
∘	`\circ`	∅	`\empty`	∪	`\union`
∩	`\intersect`	∈	`\in`	⇂	`\downleftharpoon`
○	`\bigcirc`	←	`\<-`	↦	`\mapsto`
⇒	`\=>`	⟹	`\==>`	⟦	`\[[`
⟧	`\]]`	α	`\a`	β	`\b`
γ	`\g`	δ	`\de`	ε	`\e`
σ	`\s`	₀	`\0`	₁	`\1`
₂	`\2`	₃	`\3`	₄	`\4`
₅	`\5`	₆	`\6`	₇	`\7`
₈	`\8`	₉	`\9`		

[7] jasmin.blanchette@ifi.lmu.de

Part I

Basics

Chapter 1

Types and Terms

We begin our journey by studying the basics of Lean, starting with terms (also called expressions) and their types.

Lean's logical foundation is a rich logic called the *calculus of inductive constructions*, which supports *dependent types*. Lean's logic is inspired by the λ-calculus and resembles typed functional programming languages such as Haskell, OCaml, and Standard ML. Even if you have not been exposed to these languages, you will recognize many of the concepts from modern languages (e.g., Python, C++11, Java 8). In a first approximation:

$$\text{Lean} = \text{functional programming} + \text{logic}$$

If your background is in mathematics, you probably already know most of the key concepts underlying functional programming, sometimes under different names. For example, the Haskell program

```
fib 0 = 0
fib 1 = 1
fib n = fib (n - 2) + fib (n - 1)
```

closely matches the mathematical definition

$$fib(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ fib(n-2) + fib(n-1) & \text{if } n \geq 2 \end{cases}$$

In this chapter, we restrict our attention to a fragment without dependent types, called *simple type theory* (or *higher-order logic*). It corresponds roughly to the simply typed λ-calculus [3] extended with an equality operator ($=$). It is an abstract, extremely simplified version of a programming language with a function-calling mechanism that prefigures functional programming.

1.1 Types

Types can be basic types such as $\mathbb{Z}$, $\mathbb{Q}$, and `Bool` or total functions $\sigma \to \tau$, where σ and τ are themselves types. Types indicate which values an expression may evaluate to. They introduce a discipline that is followed somewhat implicitly in mathematics. In principle, nothing prevents a mathematician from stating $1 \in 2$, but a typing discipline would mark this as the error it likely is.

Semantically, types can be viewed as sets. We would normally define the types $\mathbb{Z}$, $\mathbb{Q}$, and `Bool` so that they faithfully capture the mathematicians' $\mathbb{Z}$ and $\mathbb{Q}$ and the computer scientists' Booleans, and similarly for the function arrow ($\to$). But despite their similarities, Lean and mathematics are distinct languages. Lean's types may be *interpreted* as sets, but they are not sets.

Higher-order types are types containing $\to$ arrows that are nested on the left of $\to$, as in the type $(\mathbb{Z} \to \mathbb{Z}) \to \mathbb{Q}$. Values of such types are functions that take other functions as arguments. Accordingly, $(\mathbb{Z} \to \mathbb{Z}) \to \mathbb{Q}$ is the type of unary functions that take a function of type $\mathbb{Z} \to \mathbb{Z}$ as argument and that return a value of type $\mathbb{Q}$.

1.2 Terms

The *terms*, or expressions, of simple type theory consist of

- *constants* `c`;
- *variables* `x`;
- *applications* `t u`;
- *anonymous functions* `fun x ↦ t` (also called λ-expressions).

Above, `t` and `u` denote arbitrary terms. We can also write `t : σ` to indicate that the term `t` has the type σ.

Let us review the kinds of terms in turn:

- A constant `c : σ` is a symbol of type σ whose meaning is fixed in the current global context. For example, an arithmetic theory might contain constants such as `0 : ℤ`, `1 : ℤ`, `abs : ℤ → ℕ`, `square : ℕ → ℕ`, and `prime : ℕ → Bool`. Constants include functions (e.g., `abs`) and predicates (e.g., `prime`).
- A variable `x : σ` is either *bound* or *free*. A bound variable refers back to the input of an anonymous function `fun x : σ ↦ t` enclosing it. By contrast, a free variable is declared in the local context—a concept that will be explained below.
- An application `t u`, where `t : σ → τ` and `u : σ`, is a term of type τ denoting the result of applying the function `t` to the argument `u`. For example, if `t` is `abs` and `u` is `0`, then the application is `abs 0`. No parentheses are needed around the argument, unless it is a complex term—e.g., `prime (abs 0)`.
- Given a term `t : τ`, an anonymous function `fun x : σ ↦ t` denotes the total function of type $\sigma \to \tau$ that maps each input value `x` of type σ to the body `t`, where `x` may occur in `t`. Thus, `fun x : ℤ ↦ square (abs x)` denotes the function that maps (the value denoted by) `0` to (the value denoted by) `square (abs 0)`, that maps `1` to `square (abs 1)`, and so on. The variable `x` is said to be bound by `fun`. Accordingly, `fun` is called a *binder*. For mathematicians, a more familiar syntax would have been `x ↦ square (abs x)`, without the `fun` keyword.

Constants and variables look syntactically similar, but they are considered different. Constants are declared globally, whereas variables are introduced locally by `fun` or some other binder.

Applications and anonymous functions mirror each other: An anonymous function "builds" a function; an application "destructs" a function. What happens if we combine the two? If we apply an argument `u` of type σ to an anonymous function

`fun x : σ ↦ t[x]`, where `t[x]` represents some term that may contain `x`, then we obtain the term `t[u]`. (This is a slightly simplified view. Actually, if `t[x]` contains some binders, the bound variables might have to be renamed to avoid capturing `u`'s free variables. In Lean, this renaming takes place automatically.) Here are a few examples of applying an anonymous function:

```
(fun n : ℕ ↦ n) 4                    yields  4
(fun n : ℕ ↦ square (square n)) 5    yields  square (square 5)
(fun y : ℤ ↦ 1) 0                    yields  1
(fun x : ℤ ↦ (fun y : ℤ ↦ x)) 1      yields  fun y : ℤ ↦ 1
```

Although our functions are unary (i.e., they take one argument), we can build *n*-ary functions by nesting `fun`s, using an ingenious technique called *currying*. For example, `fun x : σ ↦ (fun y : τ ↦ x)` denotes the function of type `σ → (τ → σ)` that takes two arguments and returns the first one. Strictly speaking, `σ → (τ → σ)` takes a single argument and returns a function, which in turn takes an argument.

Applications work in the same way: If `K := (fun x : ℤ ↦ (fun y : ℤ ↦ x))`, then `K 1 = (fun y : ℤ ↦ 1)` and `(K 1) 0 = 1`. The function `K` in `K 1`, which is applied to a single argument, is said to be *partially applied*, because it can take an additional argument.

Currying is so useful a concept that we will omit most parentheses, writing

```
σ → τ → υ                for  σ → (τ → υ)
t u v                    for  (t u) v
fun x : σ ↦ fun y : τ ↦ t  for  fun x : σ ↦ (fun y : τ ↦ t)
```

and also

```
fun (x : σ) (y : τ) ↦ t  for  fun x : σ ↦ fun y : τ ↦ t
fun x y : σ ↦ t          for  fun (x : σ) (y : σ) ↦ t
```

Moreover, in an anonymous function `fun x : σ ↦ t`, we can usually omit the type annotation `: σ`, writing

```
fun x ↦ t   for  fun x : σ ↦ t
```

Lean will then try to infer the type based on the body `t` and on the context surrounding the anonymous function. Type inference lightens notations and saves some keystrokes.

In mathematics, it is customary to write binary operators in infix syntax (e.g., `x + y`). Such notations are also possible in Lean, as syntactic sugar for curried function application (e.g., `add x y`).

One way to work with Lean is to declare the types and constants we need using the `opaque` command. Consider the following declarations:

```
opaque a : ℤ
opaque b : ℤ
opaque f : ℤ → ℤ
opaque g : ℤ → ℤ → ℤ

#check fun x : ℤ ↦ g (f (g a x)) (g x b)
#check fun x ↦ g (f (g a x)) (g x b)
```

The first four lines declare four constants (`a`, `b`, `f`, `g`), which can be used to form terms. The last two lines use the `#check` command to type-check terms and show their types. Using traditional mathematical notations, the term on the last line would be written $x \mapsto g(f(g(a, x)), g(x, b))$. The `#` prefix identifies diagnosis commands: commands that are useful for debugging but that we would normally not keep in our Lean files.

1.3 Type Checking and Type Inference

When Lean parses a term, it checks whether the term is well typed. In the process, it tries to infer the types of bound variables if those are omitted—e.g., the type of `x` in `fun x ↦ square (square x)`.

For simple type theory, type checking and type inference are decidable problems. Advanced features such as overloading (the possibility to reuse the same name for several constants—e.g., `0 : ℕ` and `0 : ℝ`) can lead to undecidability [26]. Lean takes a pragmatic, computer-science-oriented approach and assumes that numerals `0`, `1`, `2`, ... are of type $\mathbb{N}$ if several types are possible.

Lean's type system can be expressed as a formal system. A *formal system* consists of *judgments* and of *derivation rules* for producing judgments. A typing judgment has the form $\mathtt{C} \vdash \mathtt{t} : \sigma$ and means that term `t` has type σ in local context `C`. For example, the judgment $\vdash \mathtt{abs} : \mathbb{Z} \to \mathbb{N}$ states that the constant `abs` has type $\mathbb{Z} \to \mathbb{N}$ in the empty local context.

The local context gives the types of the variables in `t` that are not bound by a `fun`. It is used to keep track of the variables bound by binders *outside* `t`. If the same variable `x` is bound multiple times, the last occurrence shadows the other ones. For example, the judgment $\mathtt{x} : \mathbb{Z}, \mathtt{y} : \mathbb{N}, \mathtt{y} : \mathbb{Z} \vdash \mathtt{y} : \mathbb{Z}$ states that the variable `y` has type $\mathbb{Z}$ in the local context $\mathtt{x} : \mathbb{Z}, \mathtt{y} : \mathbb{N}, \mathtt{y} : \mathbb{Z}$.

For simple type theory, the typing judgments are produced by four typing rules, one per kind of term:

$$\frac{}{\mathtt{C} \vdash \mathtt{c} : \sigma}\ \textsc{Cst} \quad \text{if } \mathtt{c} \text{ is globally declared with type } \sigma$$

$$\frac{}{\mathtt{C} \vdash \mathtt{x} : \sigma}\ \textsc{Var} \quad \text{if } \mathtt{x} : \sigma \text{ is the rightmost occurrence of } \mathtt{x} \text{ in } \mathtt{C}$$

$$\frac{\mathtt{C} \vdash \mathtt{t} : \sigma \to \tau \qquad \mathtt{C} \vdash \mathtt{u} : \sigma}{\mathtt{C} \vdash \mathtt{t}\ \mathtt{u} : \tau}\ \textsc{App}$$

$$\frac{\mathtt{C}, \mathtt{x} : \sigma \vdash \mathtt{t} : \tau}{\mathtt{C} \vdash (\mathtt{fun}\ \mathtt{x} : \sigma \mapsto \mathtt{t}) : \sigma \to \tau}\ \textsc{Fun}$$

Each rule has zero or more premises (above the horizontal bar), a conclusion (below the bar), and possibly a side condition (on the right). The premises are typing judgments, whereas the side conditions are arbitrary mathematical conditions on the mathematical variables occurring in the rule. To show the premises, we need to continue performing a derivation upward, as we will see in a moment. As for the side conditions, we can use the entire arsenal of mathematics to show that they are true.

The first two rules, labeled Cst and Var, have no premises, but they have side conditions that must be satisfied for the rules to apply. The last two rules take one

or two judgments as premises and produce a new judgment. FUN is the only rule that modifies the local context: As we enter the body `t` of an anonymous function, we need to record the existence of the bound variable `x` and its type to be ready when we meet `x` in `t`.

We can use this rule system to prove that a given term is well typed by working our way backwards (i.e., upwards) and applying the rules, building a *formal derivation* of a typing judgment. Like natural trees, derivation trees are drawn with the root at the bottom. The derived judgment appears at the root, and each branch ends with the application of a premise-less rule. Rule applications are indicated by a horizontal bar and a label. The following typing derivation establishes that the term `fun x : ℤ ↦ abs x` has type $\mathbb{Z} \to \mathbb{N}$ in an arbitrary local context `C`:

$$\dfrac{\dfrac{\dfrac{}{\texttt{C, x : } \mathbb{Z} \texttt{ } \vdash \texttt{ abs : } \mathbb{Z} \to \mathbb{N}}\,\textsc{Cst} \qquad \dfrac{}{\texttt{C, x : } \mathbb{Z} \texttt{ } \vdash \texttt{ x : } \mathbb{Z}}\,\textsc{Var}}{\texttt{C, x : } \mathbb{Z} \texttt{ } \vdash \texttt{ abs x : } \mathbb{N}}\,\textsc{App}}{\texttt{C } \vdash \texttt{ (fun x : } \mathbb{Z} \mapsto \texttt{abs x) : } \mathbb{Z} \to \mathbb{N}}\,\textsc{Fun}$$

Reading the proof from the root upwards, notice how the local context is threaded through and how it is extended by the FUN rule. The rule moves the variable bound by `fun` to the local context, making an application of VAR possible further up the tree. If the variable `x` is already declared in `C`, it becomes shadowed by `x : ℤ` after entering the `fun` expression.

In summary, the type system consists of derivation rules that can be (1) instantiated with arbitrary values for their mathematical variables and (2) connected to form derivation trees.

Here is a second example, this time starting with an empty local context:

$$\dfrac{\dfrac{\dfrac{}{\texttt{x : } \mathbb{Z}\texttt{, y : } \mathbb{Z} \texttt{ : } \vdash \texttt{ x : } \mathbb{Z}}\,\textsc{Var}}{\texttt{x : } \mathbb{Z} \vdash \texttt{ (fun y : } \mathbb{Z} \mapsto \texttt{x) : } \mathbb{Z} \to \mathbb{Z}}\,\textsc{Fun}}{\vdash \texttt{ (fun x : } \mathbb{Z} \mapsto \texttt{fun y : } \mathbb{Z} \mapsto \texttt{x) : } \mathbb{Z} \to \mathbb{Z} \to \mathbb{Z}}\,\textsc{Fun}$$

Both examples so far were well typed. If we start with an ill-typed term, or if we specify the wrong type or context in the judgment at the root of the derivation tree, we will find that we cannot complete the derivation. A derivation constitutes a proof that a term is well typed and has the specified type in the given context.

The above type system checks only that terms are well typed. It does not check that types are well formed. For example, given the unary type constructor `List`, `List` $\mathbb{Z}$ (the type of lists of integers) is well formed, whereas $\mathbb{Z}$ `List` and `List List` are ill-formed. For simple type theory, well-formedness is easy to check: Only declared type constructors should be used, and each *n*-ary type constructor should be passed exactly *n* type arguments.

1.4 Type Inhabitation

Given a type σ, the type inhabitation problem consists of finding an "inhabitant" of that type—a term of type σ—within the empty local context. It may seem like a pointless exercise, but as we will see in Chapter 4, this problem is closely related

to that of finding a proof of a proposition. Seemingly silly exercises of the form "find a term of type σ" are good practice towards mastery of theorem proving.

Although this problem is in general undecidable, with the right strategy we can go a long way. To create a term of a given type, start with the placeholder `_` and recursively apply a combination of the following two steps:

1. If the type is of the form $\sigma \to \tau$, a possible inhabitant is an anonymous function, of the form `fun x : σ ↦ _`, where `_` is a placeholder for a missing term of type τ. Lean will mark `_` as an error; if you hover over it in Visual Studio Code, a tooltip will show the missing term's type as well as any variables declared in the local context.
2. Given a type σ (which may be a function type), you can use any constant `c` or variable `x` of type $\tau_1 \to \cdots \to \tau_n \to \sigma$ to build a term of type σ. For each argument, you need to put a placeholder, yielding `c _ ... _` or `x _ ... _`.

The placeholders can be eliminated recursively using the same strategy.

As an example, we will apply the strategy to find a term of type

$$(\alpha \to \beta \to \gamma) \to ((\beta \to \alpha) \to \beta) \to \alpha \to \gamma$$

Initially, only step 1 is applicable, with

$$\sigma := \alpha \to \beta \to \gamma \quad \text{and} \quad \tau := ((\beta \to \alpha) \to \beta) \to \alpha \to \gamma$$

(Recall that $\to$ is right-associative.) This results in the term `fun f ↦ _`, which has the right type but has a placeholder left. Since the argument `f` has type σ, a function type, it makes sense to use the name `f` for it. Then we continue recursively with the placeholder, of type τ. Again, only step 1 is possible, so we end up with the term `fun f g ↦ _`, where `g` has type $(\beta \to \alpha) \to \beta$ and the placeholder has type $\alpha \to \gamma$. A third application of step 1 yields `fun f g a ↦ _`, where `a` has type α and the placeholder has type γ.

At this point, step 1 no longer applies. Let us see if step 2 is possible. The context surrounding the placeholder contains the following variables:

$$\mathtt{f} : \alpha \to \beta \to \gamma,\ \mathtt{g} : (\beta \to \alpha) \to \beta,\ \mathtt{a} : \alpha$$

Recall that we are trying to build a term of type γ. The only variable we can use to achieve this is `f`: It takes two arguments and returns a value of type γ. So we replace the placeholder with the term `f _ _`, where the two new placeholders stand for the two missing arguments. Putting everything together, we now have the term `fun f g a ↦ f _ _`.

Following `f`'s type, the placeholders are of type α and β, respectively. The first placeholder is easy to fill in, using step 2 again, by simply supplying `a`, of type α, with no arguments. For the second placeholder, we apply step 2 with the variable `g`, which is the only source of βs. Since `g` takes an argument, we must supply a placeholder. This means our current term is `fun f g a ↦ f a (g _)`.

We are almost done. The only placeholder left has type $\beta \to \alpha$, which is `g`'s argument type. Applying step 1, we replace the placeholder with `fun b ↦ _`, where `_` has type α. Here, we can simply supply `a`. Our final term is `fun f g a ↦ f a (g (fun b ↦ a))`.

The above derivation was tedious but deterministic: At each point, either step 1 or 2 was applicable, but not both. In general, this will not always be the case.

For some other types, we might encounter dead ends and need to backtrack. We might also fail altogether, with nowhere to backtrack to.

The key idea is that the term should be syntactically correct at all times. The only red underlining we should see in Visual Studio Code should appear under the placeholders. In general, a good principle for software development is to start with a program that compiles, perform the smallest change possible to obtain a new compiling program, and repeat until the program is complete.

1.5 Summary of New Lean Constructs

At the end of this and most other chapters, we include a brief summary of the constructs introduced in the chapter. Some syntaxes have multiple meanings, which will be introduced gradually. We refer to *The Lean 4 Manual* [1], the *Theorem Proving in Lean 4* [13] tutorial, and the `mathlib` documentation[1] for details.

Diagnosis Command

`#check`	checks and prints type of a term

Declaration

`opaque`	declares an unspecified new constant or type

[1] `https://leanprover-community.github.io/mathlib4_docs/`

Chapter 2

Programs and Theorems

We continue our study of the basics of Lean, focusing on programs and theorems, without carrying out any proofs yet. We review how to define new types and functions and how to state their intended properties as theorems.

2.1 Type Definitions

A distinguishing feature of Lean's calculus of inductive constructions is its built-in support for inductive types. An *inductive type* is a type whose values are built by applying special constants called *constructors*. Inductive types are a concise way of representing acyclic data in a program. You may know them under some other, largely synonymous names, including algebraic data types, inductive data types, freely generated data types, recursive data types, and data types.

2.1.1 Definition of Natural Numbers

The "Hello, World!" example of inductive types is the type `Nat` (= $\mathbb{N}$) of natural numbers. In Lean, it can be defined as follows:

```
inductive Nat : Type where
  | zero : Nat
  | succ : Nat → Nat
```

The first line announces to the world that we are introducing a new type called `Nat`, intended to represent the natural numbers. The second and third line declare two new constructors, `Nat.zero : Nat` and `Nat.succ : Nat → Nat`, that can be used to build values of type `Nat`. Following an established convention in computer science and logic, counting starts at zero. The second constructor is what makes this inductive definition interesting—it requires an argument of type `Nat` to produce a value of type `Nat`. The terms

```
Nat.zero
Nat.succ Nat.zero
Nat.succ (Nat.succ Nat.zero)
⋮
```

denote the different values of type `Nat`—zero, its successor, its successor's successor, and so on. This notation is called *unary*—or *Peano*, after the logician

Giuseppe Peano. For an alternative explanation of Peano numbers in Lean (and some groovy video game graphics), see Kevin Buzzard's article "Can computers prove theorems?"[1]

The general format of type declarations is

```
inductive type-name (params₁ : type₁) ... (paramsₖ : typeₖ) : Type
  where
  | constructor-name₁ : constructor-type₁
      ⋮
  | constructor-nameₙ : constructor-typeₙ
```

For the natural numbers, it is possible to convince Lean to use the familiar names `0`, `1`, `2`, ..., and indeed the predefined ℕ type offers such syntactic sugar. But using the lengthier notations sheds more light on Lean's type definitions. (Besides, syntactic sugar is known to cause cancer of the semicolon [25].)

In the Lean file accompanying this chapter, the definition of `Nat` is located in a namespace, delimited by `namespace MyNat` and `end MyNat`, to keep its effects contained to a portion of the file. After `end MyNat`, any occurrence of `Nat`, `Nat.zero`, or `Nat.succ` will refer to Lean's predefined type of natural numbers. Similarly, the entire file is put in the `LoVe` namespace to prevent name clashes with existing Lean libraries.

We can inspect an earlier definition at any point in Lean by using the `#print` command. For example, `#print Nat` within the `MyNat` namespace displays the following information:

```
inductive LoVe.MyNat.Nat : Type
number of parameters: 0
constructors:
LoVe.MyNat.Nat.zero : Nat
LoVe.MyNat.Nat.succ : Nat → Nat
```

The focus on natural numbers is one of the many features of this guide that reveal a bias towards computer science. Number theorists would be more interested in the integers ℤ and the rational numbers ℚ; analysts would want to work with the real numbers ℝ and the complex numbers ℂ. But the natural numbers are ubiquitous in computer science and enjoy a very simple definition as an inductive type. They can also be used to build other types, as we will see in Chapter 12.

2.1.2 Definition of Arithmetic Expressions

If we were to specify a calculator program or a programming language, we would likely need to define a type to represent arithmetic expressions as an abstract syntax tree. The next example shows how this could be done in Lean:

```
inductive AExp : Type where
  | num : ℤ → AExp
  | var : String → AExp
  | add : AExp → AExp → AExp
  | sub : AExp → AExp → AExp
  | mul : AExp → AExp → AExp
```

[1] http://chalkdustmagazine.com/features/can-computers-prove-theorems/

```
  | div : AExp → AExp → AExp
```

Mathematically, this definition is equivalent to defining the type `AExp` inductively by the following formation rules:

1. For every integer `i`, the term `AExp.num i` is an `AExp` value. (Intuitively, the constructor `AExp.num` "boxes" an integer into an arithmetic expression.)
2. For every character string `x`, the term `AExp.var x` is an `AExp` value.
3. If e_1 and e_2 are `AExp` values, then so are `AExp.add` e_1 e_2, `AExp.sub` e_1 e_2, `AExp.mul` e_1 e_2, and `AExp.div` e_1 e_2.

The above definition is exhaustive. The only possible values for `AExp` are those built using formation rules 1 to 3. Moreover, `AExp` values built using different formation rules are distinct. These two properties of inductive types are captured by the motto "No junk, no confusion" due to Joseph Goguen.

It may be instructive to compare the concise Lean specification of `AExp` above with a Java program that achieves the same. The program consists of one interface and six classes that implement it, corresponding to the `AExp` type and its six constructors:

```
public interface AExp { }

public class Num implements AExp {
    public int num;

    public Num(int num) { this.num = num; }
}

public class Var implements AExp {
    public String var;

    public Var(String var) { this.var = var; }
}

public class Add implements AExp {
    public AExp left;
    public AExp right;

    public Add(AExp left, AExp right)
    { this.left = left; this.right = right; }
}

public class Sub implements AExp {
    public AExp left;
    public AExp right;

    public Sub(AExp left, AExp right)
    { this.left = left; this.right = right; }
}

public class Mul implements AExp {
```

```
    public AExp left;
    public AExp right;

    public Mul(AExp left, AExp right)
    { this.left = left; this.right = right; }
}

public class Div implements AExp {
    public AExp left;
    public AExp right;

    public Div(AExp left, AExp right)
    { this.left = left; this.right = right; }
}
```

In C, the natural counterpart of an inductive type is a tagged union. The type declarations would be as follows:

```
#include <stddef.h>
#include <stdlib.h>

enum AExpKind {
    AEK_NUM, AEK_VAR, AEK_ADD, AEK_SUB, AEK_MUL, AEK_DIV
};

struct aexp;

struct aexp_num {
    int num;
};

struct aexp_var {
    char var[1024];
};

struct aexp_binop {
    struct aexp *left;
    struct aexp *right;
};


struct aexp {
    enum AExpKind kind;
    union {
        struct aexp_num anum;
        struct aexp_var avar;
        struct aexp_binop abinop;
    } data;
};
```

Corresponding to each constructor in Lean, we would need to write a function to allocate an `aexp` object of the right size in memory. Here is the definition of the function corresponding to the first constructor, `AExp.num`:

```
struct aexp *create_num(int num)
{
    struct aexp *res = malloc(offsetof(struct aexp, data) +
            sizeof(struct aexp_num));

    res->kind = AEK_NUM;
    res->data.anum.num = num;
    return res;
}
```

The subtle pointer arithmetic for the `malloc` call is needed to allocate exactly the right amount of memory.

2.1.3 Definition of Lists

The next type we consider is that of finite lists:

```
inductive List (α : Type) where
  | nil  : List α
  | cons : α → List α → List α
```

The type is *polymorphic*: It is parameterized by a type α, which we can instantiate with concrete types. For example, `List` $\mathbb{Z}$ is the type of lists over integers, and `List (List` $\mathbb{R}$`)` is the type of lists of lists of real numbers. The type constructor `List` takes a type as argument and returns a type. Like generics in Java and templates in C++, polymorphism is a mechanism that provides parameterized types.

The following commands display the constructors' types:

```
#check List.nil
#check List.cons
```

The output looks different from what we might expect:

```
List.nil : List ?m.2396
List.cons : ?m.2397 → List ?m.2397 → List ?m.2397
```

Even if we try to restrict ourselves to a fragment of Lean's language, Lean often exposes us to more advanced constructs in the output, such as `?m.2396` and `?m.2397` above. Our advice is to adopt a *sporty attitude*: Do not worry if you do not always understand everything the first time. Use your common sense and your imagination. And, above all, do not hesitate to ask.

Lean's built-in lists offer syntactic sugar for writing lists:

`[]`	for	`List.nil`
`x :: xs`	for	`List.cons x xs`
`[`x_1`, ..., `x_n`]`	for	x_1 `:: ... ::` x_n `:: []`

The `::` operator, like all other binary operators, binds less tightly than function application. Thus, `f x :: List.reverse ys` is parsed as `(f x) :: (List.reverse ys)`. It is good practice to avoid needless parentheses. They can quickly impair

readability. In addition, it is important to put spaces around infix operators, to suggest the right precedence.

Functional programmers often use names such as `xs`, `ys`, `zs` for lists, although `l` is also common in Lean. A list contains many elements, so a plural form is natural. A list of cats might be called `cats`; a list of lists of cats, `catss`. When a nonempty list is presented as a head and a tail, we usually write, say, `x :: xs` or `cat :: cats`.

2.2 Function Definitions

If all we want is to declare a function, we can use the `opaque` command (Section 1.2). But usually, we want to *define* the function's behavior, and for this we can use the `def` command. Because Lean's roots are in functional programming, functions are defined as mathematical expressions evaluating to a value, not as an imperative program modifying some state. Accordingly, recursion, rather than iteration with `while` or `for`, is the primary mechanism for traversing data.

2.2.1 Recursion on Natural Numbers

Let us see on a simple example how recursion works. The Lean definition of Fibonacci numbers is as follows:

```
def fib : ℕ → ℕ
  | 0     => 0
  | 1     => 1
  | n + 2 => fib (n + 1) + fib n
```

The patterns on the left correspond to the argument of the function. Here, `fib` is declared with a single argument of type ℕ, so we *pattern-match* on a single natural number. A pattern is triggered when the input has the given form. For example, if the input is `1`, then the pattern `1` matches it, and the corresponding right-hand side is evaluated. If the input is `5`, then the pattern `n + 2` is triggered, with `n := 3`. Then the right-hand side is evaluated with `n` set to `3`, yielding `fib 4 + fib 3`.

The general format of recursive definitions is

```
def name (params₁ : type₁) ... (paramsₘ : typeₘ) : type
  | patterns₁ => result₁
    ⋮
  | patternsₙ => resultₙ
```

The parameters $params_1$ to $params_m$ cannot be subjected to pattern matching, only the remaining arguments declared in *type*. If multiple patterns are provided on one line, they are separated by commas. Patterns may contain variables, which are then visible in the corresponding right-hand side, as well as constructors.

The basic arithmetic operations on natural numbers, such as addition, multiplication, and exponentiation, can be defined by recursion. Of course, they are already defined in Lean (as `+`, `*`, and `^`), but they are useful exercises. We start with addition:

```
def add : ℕ → ℕ → ℕ
  | m, Nat.zero   => m
  | m, Nat.succ n => Nat.succ (add m n)
```

We pattern-match on two arguments at the same time, distinguishing the case where the second argument is zero and the case where it is nonzero. Each recursive call to `add` peels off one `Nat.succ` constructor from the second argument. Instead of `Nat.zero` and `Nat.succ n`, Lean also allows us to write `0` and `n + 1` as syntactic sugar.

We can evaluate the result of applying `add` to numbers using `#eval` or `#reduce`:

```
#eval add 2 7
#reduce add 2 7
```

Both commands print `9`, as expected (at least in Visual Studio Code). `#eval` employs an optimized interpreter, whereas `#reduce` uses Lean's inference kernel, which is less efficient.

It is good practice to provide a few tests each time we define a function, to ensure that it behaves as expected. You can even leave the `#eval` or `#reduce` calls in your Lean files as documentation.

The definition of multiplication is similar to that of addition:

```
def mul : ℕ → ℕ → ℕ
  | _, Nat.zero   => Nat.zero
  | m, Nat.succ n => add m (mul m n)
```

The underscore (_) stands for an unused variable. We could have put a name (e.g., `m`), but _ documents our intentions better.

The `#eval` command below prints `14`, as expected:

```
#eval mul 2 7
```

Exponentiation ("*m* to the power of *n*") can be defined in various ways. Our first proposal is structurally identical to the definition of multiplication:

```
def power : ℕ → ℕ → ℕ
  | _, Nat.zero   => 1
  | m, Nat.succ n => mul m (power m n)
```

Since the first argument, `m`, remains unchanged in the recursive call, `power m n`, we can factor it out and put it next to the function's name, as a *parameter*, before the colon introducing the type of the function (excluding the parameter `m`):

```
def powerParam (m : ℕ) : ℕ → ℕ
  | Nat.zero   => 1
  | Nat.succ n => mul m (powerParam m n)
```

From the outside, there is no difference between the two definitions. In fact, we already saw this syntax for the type argument `α` of the `List` constructor (Section 2.1).

Yet another definition is possible by first introducing a general-purpose iterator and then using it with the right arguments:

```
def iter (α : Type) (z : α) (f : α → α) : ℕ → α
  | Nat.zero   => z
  | Nat.succ n => f (iter α z f n)

def powerIter (m n : ℕ) : ℕ :=
  iter ℕ 1 (mul m) n
```

The `iter` function takes a type α, a "zero" value `z`, a unary function `f`, and a natural number `n`, and computes

$$\underbrace{\texttt{f}\ (\texttt{f}\ (\cdots\ (\texttt{f}}_{\texttt{n}\text{ times}}\ \texttt{z})\ \cdots))$$

It is an example of a higher-order function: a function that takes another function (here, `f`) as argument. In functional programming, functions are treated like any other objects and can be used as arguments or as the result of a function. Here, we use `iter` to compute `m` to the power of `n` as

$$\underbrace{\texttt{mul m}\ (\texttt{mul m}\ (\cdots\ (\texttt{mul m}}_{\texttt{n}\text{ times}}\ 1)\ \cdots))$$

Finally, notice that the definition of `powerIter` is not recursive. For nonrecursive functions without pattern matching, the syntax is simply

```
def name (params₁ : type₁) ... (paramsₘ : typeₘ) : type :=
  result
```

2.2.2 Recursion on Arithmetic Expressions

Going back to the type of arithmetic expressions, if we wanted to implement an `eval` function in Java, we would probably add it as part of `AExp`'s interface and implement it in each subclass. For `Add`, `Sub`, `Mul`, and `Div`, we would recursively call `eval` on the `left` and `right` objects.

In Lean, the syntax is very compact. We define a single function and use pattern matching to distinguish the six cases:

```
def eval (env : String → ℤ) : AExp → ℤ
  | AExp.num i     => i
  | AExp.var x     => env x
  | AExp.add e₁ e₂ => eval env e₁ + eval env e₂
  | AExp.sub e₁ e₂ => eval env e₁ - eval env e₂
  | AExp.mul e₁ e₂ => eval env e₁ * eval env e₂
  | AExp.div e₁ e₂ => eval env e₁ / eval env e₂
```

Notice that this function is higher-order: It takes a function, `env`, that represents an "environment" assigning values to the variables. The environment is used in the `AExp.var` case to evaluate a variable and is threaded through in the recursive cases (`AExp.add`, `AExp.sub`, `AExp.mul`, and `AExp.div`).

Perhaps you are worried about division by zero in the `AExp.div` case. Let us see what `#eval` has to say about it, using some environment that assigns 7 to all variables:

```
#eval eval (fun x ↦ 7) (AExp.div (AExp.var "y") (AExp.num 0))
```

The output is `0`. In Lean, division is conveniently defined as a total function that returns zero when the denominator is zero. For a lucid explanation of why this is not dangerous, see Buzzard's blog.[2]

[2] https://xenaproject.wordpress.com/2020/07/05/division-by-zero-in-type-theory-a-faq/

2.2.3 Recursion on Lists

Recursive functions on lists can be defined in a similar way:

```
def append (α : Type) : List α → List α → List α
  | List.nil,       ys => ys
  | List.cons x xs, ys => List.cons x (append α xs ys)
```

The `append` function takes three arguments: a type α and two lists of type `List` α.

```
#eval append ℕ [3, 1] [4, 1, 5]
#eval append _ [3, 1] [4, 1, 5]
```

By passing the placeholder _, we leave it to Lean to infer the type $\mathbb{N}$ from the type of the other two arguments.

To make the type argument α implicit, we can put it in curly braces { }:

```
def appendImplicit {α : Type} : List α → List α → List α
  | List.nil,       ys => ys
  | List.cons x xs, ys => List.cons x (appendImplicit xs ys)

#eval appendImplicit [3, 1] [4, 1, 5]
```

The at sign (@) can be used to make the implicit arguments explicit. This is occasionally necessary to guide Lean's parser.

```
#eval @appendImplicit ℕ [3, 1] [4, 1, 5]
#eval @appendImplicit _ [3, 1] [4, 1, 5]
```

We can use syntactic sugar in the definition, both in the patterns on the left-hand sides of `=>` and in the right-hand sides:

```
def appendPretty {α : Type} : List α → List α → List α
  | [],      ys => ys
  | x :: xs, ys => x :: appendPretty xs ys
```

In Lean's standard library, the append function is an infix operator called `++`. We can use it to define a function that reverses a list:

```
def reverse {α : Type} : List α → List α
  | []      => []
  | x :: xs => reverse xs ++ [x]
```

2.3 Theorem Statements

What makes Lean a proof assistant and not only a programming language is that we can state theorems about the types and constants we define and prove that they hold. In interactive theorem proving, the terms theorem, lemma, corollary, fact, property, and true statement are used more or less interchangeably. Similarly, propositions, logical formulas, and statements will all mean the same.

In Lean, propositions are simply terms of type `Prop`. This stands in contrast with first-order logic, where terms and formulas are traditionally distinguished in the syntax. A proposition that can be proved is called a theorem (or lemma, corollary, etc.); otherwise it is a nontheorem or false statement. Mathematicians

sometimes use the term "proposition" as a synonym for theorem (e.g., "Proposition 3.14"), but in formal logic propositions can also be false.

Here are examples of true statements that can be made about the addition, multiplication, and list reversal operations defined in Section 2.2:

```
theorem add_comm (m n : ℕ) :
  add m n = add n m :=
  sorry

theorem add_assoc (l m n : ℕ) :
  add (add l m) n = add l (add m n) :=
  sorry

theorem mul_comm (m n : ℕ) :
  mul m n = mul n m :=
  sorry

theorem mul_assoc (l m n : ℕ) :
  mul (mul l m) n = mul l (mul m n) :=
  sorry

theorem mul_add (l m n : ℕ) :
  mul l (add m n) = add (mul l m) (mul l n) :=
  sorry

theorem reverse_reverse {α : Type} (xs : List α) :
  reverse (reverse xs) = xs :=
  sorry
```

The general format is

```
theorem name (params₁ : type₁) ... (paramsₘ : typeₘ) :
  statement :=
  proof
```

The `:=` symbol separates the theorem's statement and its proof. The syntax of `theorem` is very similar to that of a `def` command without pattern matching, with *`statement`* instead of *`type`* and *`proof`* instead of *`result`*. In the examples above, we put the marker `sorry` as a placeholder for the actual proof. The marker is quite literally an apology to future readers and to Lean for the absence of a proof. It is also a reason to *worry*, until we manage to eliminate it. In Chapters 3 and 4, we will see how to achieve this.

The intuitive semantic of a `theorem` command with a `sorry` proof is, "This proposition should be provable, but I have not carried out the proof yet—sorry." Sometimes, we want to express a related idea, namely, "Let us assume this proposition holds." Lean provides the `axiom` command for this, which is often used in conjunction with `opaque`. For example:

```
opaque a : ℤ
opaque b : ℤ
```

```
axiom a_less_b :
  a < b
```

After the `opaque` commands, we have no information about `a` and `b` beyond their type. The axiom specifies the desired property about them. The general format of the command is

```
axiom name (params₁ : type₁) ... (paramsₘ : typeₘ) :
  statement
```

Axioms are dangerous, because they can lead to an inconsistent logic, in which we can derive `False`. For example, if we added a second axiom stating that `a > b`, we could easily derive `False`. The history of interactive theorem proving is paved with inconsistent axioms. One anecdote among many: At the 2020 edition of the Certified Programs and Proofs conference, a submitted paper was rejected due to a flawed axiom, from which one of the peer reviewers derived `False`.[3] Therefore, we will generally avoid axioms.

From Lean's point of view, a theorem with a `sorry` proof is effectively an axiom and can be used to play havoc with the logic's consistency. To prevent misunderstandings, it is better to use `sorry` only as a temporary measure, while developing a proof, and not as an alternative to the more explicit, and honest, `axiom`.

2.4 Summary of New Lean Constructs

Diagnosis Commands

`#eval`	executes a term using an optimized interpreter
`#print`	prints the definition of a constant
`#reduce`	executes a term using Lean's inference kernel

Declarations

`axiom`	states an axiom
`def`	defines a new constant
`inductive`	introduces a type and its constructors
`namespace ... end`	collects declarations in a named scope
`theorem`	states a theorem and its proof

Proof Commands

`sorry`	stands for a missing proof or definition

[3] One of the authors of that submission reported that the axiom has now been revised and derived as a theorem. The entire formal development is now axiom-free. The revised paper has been accepted at the conference Computer Aided Verification 2020 [8].

Chapter 3

Backward Proofs

In this chapter, we see how to prove Lean theorems using tactics, and we review the most important Lean tactics. A *tactic* is a procedure that operates on the goal—the proposition to prove—and either fully proves it or produces new subgoals (or fails). When we state a theorem, the theorem statement is the initial goal. A proof is complete once all (sub)goals have been eliminated using suitable tactics. Most of the tactics described here are documented in more detail in Chapter 5 of *Theorem Proving in Lean 4* [13].

Tactics are a *backward* proof mechanism. They start from the goal and work backwards towards the already proved theorems. Consider the theorems a, $a \to b$, and $b \to c$ as well as the goal $\vdash c$. (Goals, like typing judgments, are identified by the symbol $\vdash$.) An informal backward proof is as follows:

> To prove c, by $b \to c$ it suffices to prove b.
>
> To prove b, by $a \to b$ it suffices to prove a.
>
> To prove a, we use a. □

The telltale sign of a backward proof is the phrase "it suffices to." Notice how we progress from one *goal* to another ($\vdash c$, $\vdash b$, $\vdash a$) until no goal is left to prove. By contrast, a *forward* proof would start from the theorem a and progress, one *theorem* at a time, towards the desired theorem c:

> From a and $a \to b$, we have b.
>
> From b and $b \to c$, we have c, as desired. □

A forward proof only manipulates theorems, not goals. We will study forward proofs more deeply in Chapter 4.

Informal proofs are sometimes written in a mixture of both styles. This is manageable as long as the backward steps are clearly identified, by such wording as "to prove …, it suffices to prove …." It may help to put the $\vdash$ sign in front of goals as a reminder that these claims have not been proved yet.

Another format for expressing proofs you may be familiar with is natural deduction. A natural deduction derivation corresponding to the above proof is given below:

$$\dfrac{\dfrac{\vdash b \to c \qquad \dfrac{\vdash a \to b \qquad \vdash a}{\vdash b}\,{\to}E}{\vdash c}\,{\to}E}{}$$

When read from top to bottom, the derivation corresponds to a forward proof; when read from bottom to top, it corresponds to a backward proof.

3.1 Tactic Mode

In Chapter 2, whenever a proof was required, we simply put a `sorry` placeholder. For a tactical proof, we will now write `by` to enter *tactic mode*. In this mode, we can invoke a sequence of tactics.

Tactics operate on the goal, which consists of the proposition `Q` that we want to prove and of a local context `C`. The local context consists of variable declarations of the form `x : σ` and hypotheses of the form `h : P`. We write `C ⊢ Q` to denote a goal, where `C` is a list of variables and hypotheses and `Q` is the goal's target.

To make things more concrete, consider the following Lean example:

```
theorem fst_of_two_props :
  ∀a b : Prop, a → b → a :=
  by
    intro a b
    intro ha hb
    apply ha
```

Note that the implication arrow `→` is right-associative; this means that `a → b → a` is the same as `a → (b → a)`. Intuitively speaking, the statement has the meaning "`a` implies that `b` implies `a`," or equivalently "`a` and `b` imply `a`."

Initially, the goal is simply the theorem statement:

```
⊢ ∀a b : Prop, a → b → a
```

The `by` keyword indicates that we enter tactic mode, in which we can specify a proof by means of tactics. The tactics then follow, one per line and with the same indentation for all.

The `intro a b` tactic tells Lean to fix two free variables, `a` and `b`, corresponding to the two bound variables of the same names. The tactic mimics how mathematicians work on paper: To prove a ∀-quantified proposition, it suffices to prove it for some arbitrary but fixed value of the bound variable. The goal becomes

```
a b : Prop ⊢ a → b → a
```

where the propositions `a` and `b` are declared in the goal's local context, on the left of the `⊢` symbol. We often use the same names for the free variables as for bound variables, as we did here, but this is not mandatory. In general, it is good practice to use unique names and avoid shadowing existing variables.

The `intro ha hb` tactic tells Lean to move the assumptions `a` and `b` to the local context and to call these hypotheses `ha` and `hb`. Indeed, to prove an implication, it suffices to take its left-hand side as hypothesis and prove its right-hand side. The goal becomes

```
a b : Prop, ha : a, hb : b ⊢ a
```

It is customary to prefix hypothesis names with `h`.

The `apply ha` tactic tells Lean to match the hypothesis `a`, called `ha`, against the goal `⊢ a`. Since `a` is syntactically equal to `a`, we have a match, and this completes the proof.

Informally, in a style reminiscent of pen-and-paper mathematics, we could write the proof as follows:

> Let a and b be propositions.
>
> Assume (ha) a and (hb) b are true.
>
> To prove a, we use hypothesis ha. □

(Mathematicians would probably use numeric tags such as (1) and (2) for the hypotheses instead of informative names.)

Going back to the Lean proof, we can avoid the `intro` invocations by declaring the variables and hypotheses as parameters of the theorem, as follows:

```
theorem fst_of_two_props_params (a b : Prop) (ha : a) (hb : b) :
  a :=
  by apply ha
```

Here is an example with multiple `apply`s in sequence:

```
theorem prop_comp (a b c : Prop) (hab : a → b) (hbc : b → c) :
  a → c :=
  by
    intro ha
    apply hbc
    apply hab
    apply ha
```

Putting on our mathematician's hat, we can verbalize the last proof as follows:

> Assume (ha) a is true.
>
> To prove c, by hypothesis hbc it suffices to prove b.
>
> To prove b, by hypothesis hab it suffices to prove a.
>
> To prove a, we use hypothesis ha. □

3.2 Basic Tactics

We already saw the `intro` and `apply` tactics. These are the staples of tactical proofs. Other basic tactics include `exact`, `assumption`, `rfl`, and `ac_rfl`. These tactics can go a long way, if we are patient enough to carry out the reasoning using them, without appealing to stronger proof automation. They can also be used to solve various logic puzzles.

Below, the thin, large square brackets $[\,]$ enclose optional syntax.

intro

```
intro [name₁ ... nameₙ]
```

The `intro` tactic moves the leading ∀-quantified variable or the leading assumption a → from the goal's target to the local context. The tactic takes as an optional argument the name to give to the variable or to the assumption in the context, overriding the default name. If multiple names are supplied (i.e., n > 1), multiple variables or assumptions are moved.

rfl

The `rfl` tactic proves goals of the form `⊢ l = r`, where the two sides `l` and `r` are syntactically equal *up to computation*. Computation means first and foremost the expansion of definitions but also the reduction of an application of an anonymous function to an argument, and more. These *conversions* have traditional names. The main conversions are listed below together with examples, in a global context containing `def double (n : ℕ) : ℕ := n + n`:

α-conversion	`(fun x ↦ f x) = (fun y ↦ f y)`
β-conversion	`(fun x ↦ f x) a = f a`
δ-conversion	`double 5 = 5 + 5`
ζ-conversion	`(let n : ℕ := 2; n + n) = 4`
η-conversion	`(fun x ↦ f x) = f`
ι-conversion	`Prod.fst (a, b) = a`

Applying conversions repeatedly as left-to-right rewrite rules is called *reduction*; applying a conversion once in reverse is called *expansion*.

Briefly, to prove `⊢ l = r`, the `rfl` tactic expands definitions in `l` and `r` and performs β-reduction and other reductions. It succeeds if `l` and `r` become syntactically identical at some point during this reduction process. Often, `rfl` succeeds where a mathematician would say "by definition."

apply

```
apply theorem-or-hypothesis
```

The `apply` tactic matches the goal's target with the conclusion of the specified theorem or hypothesis and adds the theorem or hypothesis's assumptions as new goals. The matching is performed up to computation.

We must invoke `apply` with care, because it can transform a provable goal into an unprovable subgoal. For example, if the goal is `⊢ True` and we `apply` the theorem `False → True`, the conclusion `True` is matched against the goal's target `True`, and we end up with the unprovable subgoal `⊢ False`. We say that `apply` is *unsafe*. In contrast, `intro` always preserves provability and is therefore *safe*.

exact

```
exact theorem-or-hypothesis
```

The `exact` tactic matches the goal's target with the specified theorem or hypothesis, closing the goal. We can often use `apply` in such situations, but `exact` communicates our intentions better.

assumption

The `assumption` tactic finds a hypothesis from the local context that matches the goal's target and `apply`s it to prove the goal.

ac_rfl

The `ac_rfl` is similar to `rfl`, but it can be used to reason up to associativity (e.g., `(a + b) + c = a + (b + c)`) and commutativity (e.g, `a + b = b + a`). This works for binary operations that are registered as associative and commutative, such as `+` and `*` on arithmetic types, and ∪ and ∩ on sets. We will see an example in Section 3.6.

sorry

The `sorry` proof command we encountered in Chapter 2 can be used at any point in a tactical proof, as a tactic. It "proves" the current goal without actually proving it. Use with care.

3.3 Reasoning about Connectives and Quantifiers

Before we learn to reason about natural numbers, lists, or other data types, we must first learn to reason about the logical connectives and quantifiers of Lean's logic. Let us start with a simple example: commutativity of conjunction (∧).

```
theorem And_swap (a b : Prop) :
  a ∧ b → b ∧ a :=
  by
    intro hab
    apply And.intro
    apply And.right
    exact hab
    apply And.left
    exact hab
```

At this point, we recommend that you move the cursor over the example in Visual Studio Code, to see the sequence of proof states. By putting the cursor on or immediately after each command, you can see the effect of the command on that line. For the last line, Lean simply reports "Goals accomplished," meaning that no subgoals remain to be proved.

The proof is a typical `intro`–`apply`–`exact` mixture. It uses the theorems

```
And.intro : ?a → ?b → ?a ∧ ?b
 And.left : ?a ∧ ?b → ?a
And.right : ?a ∧ ?b → ?b
```

where the question marks (`?`) indicate variables that can be instantiated—for example, by matching the goal's target against the conclusion of a theorem. These are called *metavariables*.

The three theorems above are the introduction rule and the two elimination rules associated with conjunction. An *introduction rule* for a logical symbol (e.g., ∧) is a theorem whose conclusion has that symbol as the outermost symbol. Dually, an *elimination rule* has the symbol in an assumption. For each logical symbol, the introduction rules tell us *how to prove* a proposition with that symbol as the outermost position. By contrast, the elimination rules tell us *how we must have proved* such a proposition.

In the above proof, we apply the introduction rule for ∧ to prove the goal ⊢ b ∧ a, and we apply the two elimination rules to extract b and a from the hypothesis a ∧ b. It may sound strange that we "eliminate" ∧ in ⊢ b ∧ a by using a so-called introduction rule. The terminology is backward because our proofs are backward.

Question marks can also arise in goals. They indicate variables that can be instantiated arbitrarily. In the middle of the proof above, right after the tactic `apply And.right`, we have the goal

```
a b : Prop, hab : a ∧ b ⊢ ?left.a ∧ b
```

where `?left.a` is a metavariable. The tactic `exact hab` matches `?left.a` (in the target) with `a` (in `hab`). The procedure used to instantiate variables to make two terms syntactically equal is called *unification*. *Matching* is a special case of unification where one of the two terms contains no variables, like here.

Incidentally, whenever metavariables appear in goals, additional subgoals will also appear, with the type of each metavariable as a subgoal (e.g., ⊢ `Prop`). These confusing subgoals are merely a reminder that we will have to instantiate the metavariables with terms of the right type. We can usually ignore these subgoals. As soon as the metavariable is instantiated (typically when we solve another subgoal), its associated subgoal will also go away.

In Lean, unification is performed up to computation. For example, the terms `(fun x ↦ ?m) a` and `b` can be unified by taking `?m := b`, because `(fun x ↦ b) a` and `b` are syntactically equal up to β-conversion.

The following is an alternative proof of the theorem `And_swap`:

```
theorem And_swap_braces :
  ∀a b : Prop, a ∧ b → b ∧ a :=
  by
    intro a b hab
    apply And.intro
    { exact And.right hab }
    { exact And.left hab }
```

The theorem is stated differently, with `a` and `b` as ∀-quantified variables instead of parameters of the theorem. Logically, this is equivalent, but in the proof we must then `introduce` `a` and `b` in addition to `hab`.

Another difference is the use of curly braces { }. When we face two or more goals to prove, it is generally good style to put each proof in its own block enclosed in curly braces. The { } tactic combinator focuses on the first subgoal; the tactic inside must prove it. In our example, the `apply And.intro` tactic creates two subgoals, ⊢ b and ⊢ a.

A third difference is that we now apply, by juxtaposition, `And.right` and `And.left` directly to the hypothesis `a ∧ b` to obtain `b` and `a`, respectively, instead of waiting for the theorems' assumptions to emerge as new subgoals. This is a small forward step in an otherwise backward proof. The same syntax is used both to discharge (i.e., prove) a hypothesis and to instantiate a ∀ quantifier. One benefit of this approach is that we avoid the potentially confusing `?left.a` metavariable.

In the next example, we use juxtaposition to instantiate a ∀ quantifier:

```
theorem f5_if (h : ∀n : ℕ, f n = n) :
  f 5 = 5 :=
  by exact h 5
```

If `h` is the theorem `∀n, f n = n`, then `h 5` is the theorem `f 5 = 5`.

The introduction and elimination rules for disjunction (`∨`) are as follows:

```
Or.inl : ∀b : Prop, ?a → ?a ∨ b
Or.inr : ∀b : Prop, ?a → b ∨ ?a
Or.elim : ?a ∨ ?b → (?a → ?c) → (?b → ?c) → ?c
```

The `∀` quantifiers in `Or.inl` ("introduction left") and `Or.inr` ("introduction right") can be instantiated directly by applying the theorem name to the value we want to instantiate with, via simple juxtaposition. Thus, `Or.inl False` corresponds to the theorem `?a → ?a ∨ False`. This is the forward style.

Alternatively, we can invoke `apply Or.inl` on a goal of the form `... ⊢ c ∨ d`. This sets `?a := c` and `?b := d` in the theorem. The new subgoal is `... ⊢ c`. This is the backward style.

Both `Or.inl` and `Or.inr` are unsafe: If you apply the wrong one of the two, or either of them too early in a proof, you might end up with an unprovable subgoal. This is easy to see if you consider the provable goal `⊢ True ∨ False`: applying `Or.inr` yields the unprovable subgoal `⊢ False`.

The `Or.elim` rule may seem counterintuitive at first glance. In essence, it states that if we have `a ∨ b`, then to prove an arbitrary `c`, it suffices to prove `c` when `a` holds and when `b` holds. You can think of `(?a → ?c) → (?b → ?c) → ?c` as a clever trick to express the meaning of disjunction using only implication.

The introduction and elimination rules for equivalence (`↔`) are as follows:

```
Iff.intro : (?a → ?b) → (?b → ?a) → (?a ↔ ?b)
   Iff.mp : (?a ↔ ?b) → ?a → ?b
  Iff.mpr : (?a ↔ ?b) → ?b → ?a
```

The introduction and elimination rules for existential quantification (`∃`) are

```
Exists.intro : ∀w, (?P w → (∃x, ?P x))
 Exists.elim : (∃x, ?P x) → (∀a, ?P a → ?c) → ?c
```

The introduction rule for `∃` can be used to instantiate an existential quantifier with a witness. For example:

```
theorem Exists_double_iden :
  ∃n : ℕ, double n = n :=
  by
    apply Exists.intro 0
    rfl
```

Again, we instantiate a `∀` quantifier in a forward fashion: `Exists.intro 0` is the theorem `?P 0 → (∃x, ?P x)`. The rule is unsafe: Choosing the wrong witness for `x` will result in an unprovable goal. For example, if the goal is `⊢ ∃n, n > 5` and we take `3` as the witness, we end up with the unprovable subgoal `⊢ 3 > 5`.

The elimination rule for `∃` is reminiscent of that for `∨`. Indeed, a fruitful way to think of a quantification `∃n, ?P n` is as a possibly infinitary disjunction `?P 0 ∨ ?P 1 ∨ ⋯`. Similarly, `∀n, ?P n` can be thought of as `?P 0 ∧ ?P 1 ∧ ⋯`.

For truth (`True`), there is only an introduction rule:

```
True.intro : True
```

Truth holds no information whatsoever. If it appears as a hypothesis, it is completely useless, and there is no elimination rule that can extract any information from it. The `clear` tactic, described in Section 3.8 below, can be used to remove such useless hypotheses.

Dually, for falsehood (`False`), there is only an elimination rule:

```
False.elim : False → ?a
```

There is no way to prove falsehood, but if we somehow have it from somewhere (e.g., from a hypothesis), then we can derive `?a`nything.

In fact, negation (`Not`) is defined in terms of implication and falsehood: `¬ a` abbreviates `a → False`. Intuitively, these mean the same. We can think of "not `a`" as saying that "`a` would imply something absurd (hence not `a`)."

Lean's logic is classical, with support for the law of excluded middle and proof by contradiction:

```
Classical.em : ∀a : Prop, a ∨ ¬ a
Classical.byContradiction : (¬ ?a → False) → ?a
```

Implication (`→`) and universal quantification (`∀`) are the proverbial dogs that did not bark. They do not have introduction or elimination rules. Instead, for both of them, the `intro` tactic is the introduction principle, and application (as in the λ-calculus) is the elimination principle. For example, given the theorems `hab : a → b` and `ha : a`, the application `hab ha` is a theorem stating `b`.

For proving logic puzzles involving connectives and quantifiers, we advocate a "mindless," "video game" style of reasoning that relies mostly on basic tactics such as `intro` and `apply`. Here are some strategies that often work:

- If the goal's target is an implication `P → Q`, invoke `intro hP` to move `P` into your hypotheses: `..., hP : P ⊢ Q`.
- If the goal's target is a universal quantification `∀x : σ, Q`, invoke `intro x` to move `x` into the local context: `..., x : σ ⊢ Q`.
- Look for a theorem or hypothesis whose conclusion has the same shape as the goal's target (possibly containing variables that can be matched), and `apply` it. For example, if the goal's target is `Q` and you have a theorem or hypothesis of the form `hPQ : P → Q`, try `apply hPQ`.
- A negated goal `⊢ ¬ P` is syntactically equal to `⊢ P → False` up to computation, so you can invoke `intro hP` to produce the subgoal `hP : P ⊢ False`. Expanding negation's definition by invoking `rw [Not]` (described in Section 3.5) is often a good strategy.
- Sometimes you can make progress by replacing the goal by `False`, by entering `apply False.elim`. As the next step, you would typically `apply` a theorem or hypothesis of the form `P → False` or `¬ P`.
- When you face several choices (e.g., between `Or.inl` and `Or.inr`), remember which choices you have made, and backtrack when you reach a dead end or feel that you are not making any progress.
- If you suspect that you might have reached a dead end, check whether the goal actually is provable under the given assumptions. Even if you started with a provable theorem statement, the current goal might be unprovable (e.g., if you applied unsafe rules).

3.4 Reasoning about Equality

Equality (=) is also a basic logical constant. It is characterized by the following introduction and elimination rules:

```
 Eq.refl : ∀a, a = a
 Eq.symm : ?a = ?b → ?b = ?a
Eq.trans : ?a = ?b → ?b = ?c → ?a = ?c
Eq.subst : ?a = ?b → ?P ?a → ?P ?b
```

The first three theorems are introduction rules specifying that = is an equivalence relation. The fourth theorem is an elimination rule that allows us to replace equals for equals in an arbitrary context, represented by the metavariable `?P`.

An example will show some of these rules in action. Below, we apply `Eq.trans` and `Eq.symm` to prove `a = c` using the equations `a = b` and `c = b`:

```
theorem Eq_trans_symm {α : Type} (a b c : α)
    (hab : a = b) (hcb : c = b) :
  a = c :=
  by
    apply Eq.trans
    { exact hab }
    { apply Eq.symm
      exact hcb }
```

Since rewriting in this way is such a common operation, Lean provides a `rw` tactic to achieve the same result. The tactic will also notice if `rfl` is applicable:

```
theorem Eq_trans_symm_rw {α : Type} (a b c : α)
    (hab : a = b) (hcb : c = b) :
  a = c :=
  by
    rw [hab]
    rw [hcb]
```

A note on parsing: Equality binds more tightly than the logical connectives. Thus, `a = b ∧ c = d` is read as `(a = b) ∧ (c = d)`.

3.5 Rewriting Tactics

The rewriting tactic `rw` and its relative `simp` replace equals for equals. They use equations as left-to-right rewrite rules, replacing occurrences of the left-hand side by the right-hand side.

By default, they operate on the goal's target, but they can also be used to rewrite hypotheses specified using the `at` keyword:

`at` $h_1 \ldots h_n$	rewrites the specified hypotheses
`at *`	rewrites all hypotheses and the target

rw

`rw` [*theorem-or-constant*$_1$, ..., *theorem-or-constant*$_n$] [`at` *position*]

The `rw` tactic rewrites the goal using one or more equations as left-to-right rewrite rules. It searches for the first subterm that matches any of the equations' left-hand side; once found, all occurrences of that subterm are replaced by the right-hand side of the equation. If the equation contains variables, these are instantiated as necessary. To use a theorem in reverse, as a right-to-left rewrite rule, put a short left arrow (`←`) in front of the theorem's name.

Thus, given the theorem `hg : ∀x, g x = f x` and the goal `⊢ h (f a) (g b) (g c)`, the tactic `rw [hg]` produces the subgoal `⊢ h (f a) (f b) (g c)`, whereas `rw [←hg]` produces the subgoal `⊢ h (g a) (g b) (g c)`.

Instead of a theorem, we can also specify the name of a constant. This will attempt to use one of the constant's defining equations as rewrite rules.

simp

```
simp [at position]
```

The `simp` tactic rewrites the goal using a standard set of rewrite rules, called the *simp set*, exhaustively. Each equation in the simp set is used as a left-to-right rewrite rule. The simp set contains various rules about predefined symbols (e.g., arithmetic and list operators) and can be extended by putting the `@[simp]` attribute on suitable theorems.

```
simp [theorem-or-constant₁, ..., theorem-or-constantₙ] [at position]
```

For the above `simp` variant, the specified theorems are temporarily added to the simp set. In the theorem list, an asterisk (`*`) can be use to represent all hypotheses. The minus sign (`-`) in front of a theorem name temporarily removes the theorem from the simp set. A powerful incantation that both simplifies the hypotheses and uses the result to simplify the goal's target is `simp [*] at *`.

Given the theorem `hg : ∀x, g x = f x` and the goal `⊢ h (f a) (g b) (g c)`, the tactic `simp [hg]` produces the subgoal `⊢ h (f a) (f b) (f c)`, where both `g b` and `g c` have been rewritten. Instead of a theorem, we can also specify the name of a constant. This temporarily adds the constant's defining equations to the simp set.

At this point, you might wonder, "So what does `simp` do exactly?" Of course, you could study the source code or look up the scientific literature. But this might not be the most efficient use of your time. In truth, even expert users of proof assistants do not fully understand the behavior of the tactics they use daily. The most successful users adopt a relaxed, sporty attitude, trying tactics in sequence and studying the emerging subgoals, if any, to see if they are on the right path.

As you keep on using `simp` and other tactics, you will develop some intuition about what kinds of goals they work well on. This is one of the many reasons why interactive theorem proving can be learned only by doing. Often, you will not understand exactly what Lean does—why a tactic succeeds, or fails. Theorem proving can be very frustrating at times. The advice printed in large, friendly letters on the cover of *The Hitchhiker's Guide to the Galaxy* applies here: DON'T PANIC.

3.6 Proofs by Mathematical Induction

The `induction` tactic performs structural induction on a value of an inductive type. *Structural induction* means that the induction follows the structure of the

inductive type. For natural numbers constructed from `Nat.zero` and `Nat.succ`, structural induction corresponds to standard mathematical induction: To prove `p n`, it suffices to prove `p 0` and `∀k, p k → p (k + 1)`. Equipped with `induction`, we can reason about the addition and multiplication operations we defined by recursion in Section 2.2.

Addition is defined by recursion on its second argument. We will prove two theorems, `add_zero` and `add_succ`, that give us alternative equations that recurse on the *first* argument. We start with `add_zero`:

```
theorem add_zero (n : ℕ) :
  add 0 n = n :=
  by
    induction n with
    | zero       => rfl
    | succ n' ih => simp [add, ih]
```

The induction tactic is followed by two subproofs identified by the constructor they correspond to. In addition, any variables or hypotheses specific to the subproof can be named explicitly after the name of the constructor.

The first case, labeled `zero`, corresponds to the base case `⊢ add 0 0 = 0`. The second case, labeled `succ`, corresponds to the induction step

```
n' : ℕ, ih : add 0 n' = n' ⊢ add 0 (Nat.succ n') = Nat.succ n'
```

The names `n'` and `ih` specified after `succ` are the names we give to the argument of `Nat.succ` and the induction hypothesis, respectively. Other names are possible, but it is generally good practice to call the induction hypothesis `ih`.

We can keep on proving theorems by structural induction:

```
theorem add_succ (m n : ℕ) :
  add (Nat.succ m) n = Nat.succ (add m n) :=
  by
    induction n with
    | zero       => rfl
    | succ n' ih => simp [add, ih]

theorem add_comm (m n : ℕ) :
  add m n = add n m :=
  by
    induction n with
    | zero       => simp [add, add_zero]
    | succ n' ih => simp [add, add_succ, ih]

theorem add_assoc (l m n : ℕ) :
  add (add l m) n = add l (add m n) :=
  by
    induction n with
    | zero       => rfl
    | succ n' ih => simp [add, ih]
```

Once we have proved that a binary operator is commutative and associative, it is a good idea to let Lean's automation, notably `ac_rfl`, know about this. The following commands achieve this for `add`:

```
instance IsAssociative_add : IsAssociative ℕ add :=
  { assoc := add_assoc }

instance IsCommutative_add : IsCommutative ℕ add :=
  { comm := add_comm }
```

(The `instance` mechanism will be explained in Chapter 5.) The following example uses the `ac_rfl` tactic to reason up to associativity and commutativity of `add`:

```
theorem mul_add (l m n : ℕ) :
  mul l (add m n) = add (mul l m) (mul l n) :=
  by
    induction n with
    | zero       => rfl
    | succ n' ih =>
      simp [add, mul, ih]
      ac_rfl
```

Here are a few hints on how to carry out proofs by induction:

- It is usually beneficial to perform induction following the structure of the definition of one of the functions appearing in the goal. In particular, if a function is defined by recursion on its *n*th argument, it can make sense to perform the induction on that argument.
- If the base case of an induction is difficult, this is often a sign that the wrong variable was chosen or that some theorems should be proved first.

3.7 Induction Tactic

induction

```
induction term [with
| constructor₁ names₁ => tactics₁
    ⋮
| constructorₙ namesₙ => tacticsₙ]
```

The `induction` tactic performs structural induction on the specified term. This gives rise to as many subgoals as there are constructors in the definition of the term's type. Induction hypotheses are available as hypotheses in the subgoals corresponding to recursive constructors (e.g., `Nat.succ` or `List.cons`). The optional names $names_1$, ..., $names_n$ are used for any emerging variables or hypotheses.

3.8 Cleanup Tactics

The following tactics help us clean up the goal. We did not need them so far, but they can be helpful during proof exploration.

clear

```
clear variable-or-hypothesis₁ ... variable-or-hypothesisₙ
```

The `clear` tactic removes the specified variables and hypotheses, as long as they are not used anywhere else in the goal.

rename

```
rename type-of-variable-or-proposition-of-hypothesis => new-name
```

The `rename` tactic changes the name of a variable or hypothesis.

3.9 Summary of New Lean Constructs

Attribute

`@[simp]`	adds a theorem to the simp set

Proof Command

`by`	initiates a tactical proof

Tactics

`ac_rfl`	proves `l = r` up to associativity and commutativity
`apply`	matches the goal's target against the theorem's conclusion
`assumption`	proves the goal using a hypothesis
`clear`	removes a variable or hypothesis from the goal
`exact`	proves the goal using the specified theorem
`induction`	performs structural induction on a variable of an inductive type
`intro`	moves ∀-quantified variables into the goal's hypotheses
`rename`	renames a variable or hypothesis
`rfl`	proves `l = r` up to computation
`rw`	rewrites once using the given theorem as a left-to-right rewrite rule
`simp`	rewrites using a set of preregistered rewrite rules exhaustively
`sorry`	stands for a missing proof

Tactic Combinator

`{ ... }`	focuses on the first subgoal; needs to prove that subgoal

Chapter 4

Forward Proofs

Tactical proofs work backwards. They start from the goal and reduce it to existing theorems and hypotheses. Often it makes sense to work in a forward fashion: to start with existing theorems and hypotheses and proceed step by step towards our goal. *Structured proofs* are a style that supports this kind of reasoning. Tactical proofs tend to be easier to write but harder to read. Most users combine the two styles, using whichever seems the most appropriate for the situation. The higher readability of structured proofs make them popular with some users.

Structured proofs are syntactic sugar sprinkled over Lean's *proof terms*. They are built using keywords such as `assume`, `have`, `let`, and `show` that mimic pen-and-paper proofs. All Lean proofs, whether tactical or structured, are reduced internally to proof terms. We have seen some specimens already, in Chapter 3: Given hypotheses `ha : a` and `hab : a → b`, the term `hab ha` is a proof term for the proposition `b`, and we write `hab ha : b`. The names of theorems and hypotheses, such as `ha` and `hab`, are also proof terms. Pushing this idea further, given `hbc : b → c`, we can build the proof term `hbc (hab ha)` for the proposition `c`. We can think of `hab` as a function that converts a proof of `a` to a proof of `b`, and similarly for `hbc`.

Structured proofs are the default in Lean. They can be used outside tactic mode. To enter tactic mode, we need to use the `by` command.

The concepts covered here are described in more detail in Chapters 2 to 4 of *Theorem Proving in Lean 4* [13]. Nederpelt and Geuvers's textbook [22] and Van Raamsdonk's lecture notes [27] are other useful references.

4.1 Structured Proofs

As a first example, consider the following structured proof:

```
theorem fst_of_two_props :
  ∀a b : Prop, a → b → a :=
  fix a b : Prop
  assume ha : a
  assume hb : b
  show a from
    ha
```

Each variable bound by a ∀ quantifier and each assumption of an implication is introduced explicitly in the proof using the `fix` and `assume` commands. Several variables can be introduced simultaneously. We will often omit the types of variables,

especially when they can be guessed from their names; however, we will always spell out the propositions and put them one per line, to increase readability—which is, after all, one of the main potential advantages of the structured style. The `show ... from` command at the end repeats the proposition to prove, for the sake of readability, and gives the proof after the keyword `from`. The goal at this point is `a b : Prop, ha : a, hb : b ⊢ a`.

Informally, we could write the proof as follows:

> Fix some propositions `a` and `b`.
> Assume (`ha`) `a` and (`hb`) `b` are true.
> We must show `a`. This follows trivially from `ha`. □

Some authors would insert qualifiers such as "arbitrary but fixed" in front of "propositions," or they would write "Let `a` and `b` be some propositions." All these variants are equivalent. And instead of "□," we could write "QED" to conclude the proof.

The Lean proof above is atypical in that the goal's target appears among the hypotheses. Usually, we must perform some intermediate reasoning steps, essentially of the form "from so-and-so, we have such-and-such." In Lean, each intermediate step takes the form of a `have` command, as in the following example:

```
theorem prop_comp (a b c : Prop) (hab : a → b) (hbc : b → c) :
  a → c :=
  assume ha : a
  have hb : b :=
    hab ha
  have hc : c :=
    hbc hb
  show c from
    hc
```

Informally:

> Assume (`ha`) `a` is true.
> From `ha` and `hab`, we have (`hb`) `b`.
> From `hb` and `hbc`, we have (`hc`) `c`.
> We must show `c`. This follows trivially from `hc`. □

Notice that this is a forward proof: It progresses one theorem at a time from the hypothesis `a` to the desired theorem `c`. Modus ponens ("from `a` and `a → b`, derive `b`") is expressed by simple juxtaposition (e.g., `hab ha`).

In general, the `fix`–`assume`–`show` skeleton repeats the theorem's statement. We often name the fixed variables after the bound variables from the goal, as we did here, but this is not mandatory. Moreover, it is good practice to avoid shadowing existing variables. Between the last `fix` or `assume` and the `show`, we can have as many `have` commands as we want, depending on how detailed we want the argument to be. Details can increase readability, but providing too many details can overwhelm the reader.

The `have` command has a similar syntax to `theorem` but appears inside a structured proof. We can also think of a `have` as a definition. In `have hb : b := hab ha`,

the right-hand side `hab ha` is a proof term for `b`, and the left-hand side `hb` is defined as a synonym for that proof term. From then on, `hb` and `hab ha` can be used interchangeably. Since `hb` and `hc` are used only once and their proofs are very short, experts would tend to inline them, replacing `hc` by `hbc hb` and then `hb` by `hab ha`, yielding

```
theorem prop_comp_inline (a b c : Prop) (hab : a → b)
  (hbc : b → c) :
  a → c :=
  assume ha : a
  show c from
    hbc (hab ha)
```

A typical structured proof has the following `fix–assume–have–show` format:

```
theorem hr :
  ∀(c₁ : σ₁) ... (cₗ : σₗ), P₁ → ··· → Pₘ → R :=
  fix (c₁ : σ₁) ... (cₗ : σₗ)
  assume h₁ : P₁
    ⋮
  assume hₘ : Pₘ
  have k₁ : Q₁ := ...
    ⋮
  have kₙ : Qₙ := ...
  show R from ...
```

4.2 Structured Constructs

The previous section presented the main commands for writing structured proofs: `fix`, `assume`, `have`, and `show`. We now review the components of structured proofs more systematically.

Theorem or Hypothesis

The simplest structured proof, apart from `sorry`, is the name of a theorem or hypothesis. If we have

```
theorem two_add_two_Eq_four :
  2 + 2 = 4 :=
  by ...
```

then the theorem name `two_add_two_Eq_four` can be used as a proof of `2 + 2 = 4` later. For example:

```
theorem this_time_with_feelings :
  2 + 2 = 4 :=
  two_add_two_Eq_four
```

We can pass arguments to theorems to instantiate ∀ quantifiers and to discharge assumptions. Suppose the theorem `add_comm (m n : ℕ) : add m n = add n m` is available, and suppose we want to prove its instance `add 0 n = add n 0`. This can be achieved neatly using the name of the theorem and two arguments:

```
theorem add_comm_zero_left (n : ℕ) :
  add 0 n = add n 0 :=
  add_comm 0 n
```

This is equivalent to the tactical proof `by exact add_comm 0 n` but is more concise. The `exact` tactic can be seen as the inverse of `by`. Why enter tactic mode only to leave it immediately?

Like with `exact` and `apply`, the theorem or hypothesis's statement is matched with the current goal up to computation. This gives some flexibility.

fix

```
fix names : type
```

The `fix` command moves ∀-quantified variables from the goal's target to the local context. It is a structured version of the `intro` tactic.

Note that the standard Lean tactic for fixing variables is `fun`. We prefer to use `fix`, which is provided by `LoVelib`, as a more readable alternative.

assume

```
assume name : proposition
```

The `assume` command moves the leading assumption from goal's target to the local context. It can be seen as a structured version of the `intro` tactic.

Note that the standard Lean tactic for stating assumptions is `fun`. We prefer to use `assume`, which is provided by `LoVelib`, as a more readable alternative.

have

```
have name : proposition :=
  proof
```

The `have` command lets us state and prove an intermediate theorem, which may refer to names introduced by previous `fix`es, `assume`s, and `have`s. The proof can be tactical or structured. Generally, we tend to use structured proofs to sketch the main argument and resort to tactical proofs for proving subgoals or uninteresting intermediate steps. Another kind of mixture arises when we pass arguments to theorem names. For example, given `hab : a → b` and `ha : a`, the tactic `exact hab ha` will prove the goal `⊢ b`. Here, `hab ha` is a proof term nested inside a tactic.

let

```
let name [: type] := term
```

The `let` command introduces a new local definition. It can be used to name a complex object that occurs several times in the proof afterwards. It is similar to `have` but is designed for computable data, not proofs. Expanding or introducing a `let` corresponds to ζ-conversion (Section 3.2).

The construct `let name [: type] := term` must be followed by a line break or a semicolon (`;`).

show

```
show proposition from
  proof
```

The `show` command lets us repeat the goal to prove, which can be useful as documentation. It also allows us to rephrase the goal in a syntactically equal form up to computation. Instead of the syntax `show` *`proposition`* `from` *`proof`*, we can simply write *`proof`* if we do not want to repeat the goal and do not need to rephrase it. The proof can be tactical or structured.

4.3 Forward Reasoning about Connectives and Quantifiers

Reasoning about the logical connectives and quantifiers in a forward fashion uses the same introduction and elimination rules as in tactic mode (Section 3.3). A few examples will give the flavor. Let us start with conjunction:

```
theorem And_swap (a b : Prop) :
  a ∧ b → b ∧ a :=
  assume hab : a ∧ b
  have ha : a :=
    And.left hab
  have hb : b :=
    And.right hab
  show b ∧ a from
    And.intro hb ha
```

Even readers who do not know what `And.left` etc. mean can understand that we extract `a` and `b` from `a ∧ b` and put them back together as `b ∧ a`. Mathematicians would probably have an easier time making sense of this proof than of its tactical counterpart:

```
theorem And_swap_tactical (a b : Prop) :
  a ∧ b → b ∧ a :=
  by
    intro hab
    apply And.intro
    apply And.right
    exact hab
    apply And.left
    exact hab
```

In general, backward proofs are easier to derive mindlessly, and most of the automation provided by proof assistants works backwards. This makes sense: Pretend that you are Miss Marple or Hercule Poirot on a murder investigation. A backward investigation would start from the crime scene and try to extract clues that potentially connect a handful of suspects to the crime. In contrast, a forward investigation might start by questioning as many as eight billion people to determine whether they have an alibi. Which approach is more likely to succeed?

Our next examples concern *one-point rules*. These are theorems that can be used to eliminate a quantifier when the bound variable can effectively take only

one value. For example, the proposition `∀n, n = 666 → beast ≥ n` can be simplified to `beast ≥ 666`. The following theorem justifies this simplification:

```
theorem Forall.one_point {α : Type} (t : α) (P : α → Prop) :
  (∀x, x = t → P x) ↔ P t :=
  Iff.intro
    (assume hall : ∀x, x = t → P x
     show P t from
       by
         apply hall t
         rfl)
    (assume hp : P t
     fix x : α
     assume heq : x = t
     show P x from
       by
         rw [heq]
         exact hp)
```

The proof may look intimidating, but it was not hard to develop. The key was to proceed one step at a time. At first, we observed that the goal is an implication, so we wrote

```
Iff.intro (_) (_)
```

The two placeholders are important to make this proof well formed. We already put parentheses because we strongly suspect that these will be nontrivial proofs. Because proofs are basically terms, which are basically programs, the advice we gave in Section 1.4 applies here as well, mutatis mutandis:

> The key idea is that the proof should be syntactically correct at all times. The only red underlining we should see in Visual Studio Code should appear under the placeholders. In general, a good principle for proof development is to start with a proof that compiles, perform the smallest change possible to obtain a new compiling proof, and repeat until the proof is complete.

Hovering over the first placeholder makes the corresponding subgoal appear. We can see that Lean expects a proof of `⊢ (∀x, x = t → P x) → P t`, so we provide a suitable skeleton. A structured proof of an implication consists of an `assume` followed by a `show`:

```
Iff.intro
  (assume hall : ∀x, x = t → P x
   show P t from
     _)
  (_)
```

Each of the remaining placeholder can be replaced by a structured proof or a tactical proof. To fill these placeholders, we can use essentially the same procedure as for exhibiting an inhabitant of a type (Section 1.4), interpreting implication as the function arrow and writing `assume`–`show` instead of `fun`.

Let us check that the rule actually works on our motivating example:

```
theorem beast_666 (beast : ℕ) :
  (∀n, n = 666 → beast ≥ n) ↔ beast ≥ 666 :=
  Forall.one_point _ _
```

It works. Matching `Forall.one_point t P` against the statement of `beast_666` yields the instantiation `t := 666` and `P := (fun m ↦ beast ≥ m)`.

Finally, the one-point rule for ∃ demonstrates how to use the introduction and elimination rules for ∃ in a structured proof:

```
theorem Exists.one_point {α : Type} (t : α) (P : α → Prop) :
  (∃x : α, x = t ∧ P x) ↔ P t :=
  Iff.intro
    (assume hex : ∃x, x = t ∧ P x
     show P t from
       Exists.elim hex
         (fix x : α
          assume hand : x = t ∧ P x
          have hxt : x = t :=
            And.left hand
          have hpx : P x :=
            And.right hand
          show P t from
            by
              rw [←hxt]
              exact hpx))
    (assume hp : P t
     show ∃x : α, x = t ∧ P x from
       Exists.intro t
         (have tt : t = t :=
            by rfl
          show t = t ∧ P t from
            And.intro tt hp))
```

Notice how we use `Exists.elim hex` to obtain an `x` such that `x = t ∧ P x`.

4.4 Calculational Proofs

In informal mathematics, we often express proofs as transitive chains of equalities, inequalities, or equivalences (e.g., `a = b = c`, `a ≥ b ≥ c`, or `a ↔ b ↔ c`). In Lean, such *calculational proofs* are supported by the `calc` command, which provides a lightweight syntax and takes care of applying transitivity theorems for preregistered relations, such as equality and the arithmetic comparison operators.

The general syntax is as follows:

```
calc
  term₀ op₁ term₁ :=
    proof₁
  _ op₂ term₂ :=
    proof₂
      ⋮
```

```
  _ op_n term_n :=
    proof_n
```

The underscores (_) are part of the syntax. Each $proof_i$ justifies the statement $term_{i-1}\ op_i\ term_i$. The operators op_i need not be identical, but they must be compatible with each other. For example, =, $<$, and $\leq$ are compatible, whereas $>$ and $<$ are not.

A simple example follows:

```
theorem two_mul_example (m n : ℕ) :
  2 * m + n = m + n + m :=
calc
  2 * m + n = m + m + n :=
    by rw [Nat.two_mul]
  _ = m + n + m :=
    by ac_rfl
```

Mathematicians (assuming they would condescend to offer a justification for such a trivial result) could have written the above proof roughly as follows:

```
2 * m + n = (m + m) + n          (since 2 * m = m + m)
        = m + n + m          (by associativity and commutativity of +)
                                                                    □
```

In the Lean proof, the underscore stands for the term `(m + m) + n`, which we would have had to repeat had we written the proof without `calc`:

```
theorem two_mul_example_have (m n : ℕ) :
  2 * m + n = m + n + m :=
  have hmul : 2 * m + n = m + m + n :=
    by rw [Nat.two_mul]
  have hcomm : m + m + n = m + n + m :=
    by ac_rfl
  show _ from
    Eq.trans hmul hcomm
```

Notice that with `have`s, we also need to explicitly invoke `Eq.trans` and to give names to the two intermediate steps.

4.5 Forward Reasoning with Tactics

Many users prefer the tactic mode to structured proofs. But even in tactic mode, it can be useful to reason in a forward fashion, mixing forward and backward reasoning steps. The structured proof commands `have`, `let`, and `calc` are also available as tactics, making this possible.

The following example demonstrates the `have` and `let` tactics on a theorem we have seen several times already:

```
theorem prop_comp_tactical (a b c : Prop) (hab : a → b)
  (hbc : b → c) :
  a → c :=
  by
```

```
intro ha
have hb : b :=
  hab ha
let c' := c
have hc : c' :=
  hbc hb
exact hc
```

have

```
have name : proposition :=
  proof
```

The `have` tactic lets us state and prove an intermediate theorem in tactic mode. Afterwards, the theorem is available as a hypothesis in the goal state.

let

```
let name [: type] := term
```

The `let` tactic lets us introduce a local definition in tactic mode. Afterwards, the defined symbol and its definition are available in the goal state.

calc

```
calc
  term₀ op₁ term₁ :=
    proof₁
  _ op₂ term₂ :=
    proof₂
      ⋮
  _ opₙ termₙ :=
    proofₙ
```

The `calc` tactic lets us enter a calculational proof (Section 4.4) in tactic mode. The tactic has the same syntax as the structured proof command of the same name. We can regard `calc ...`, where `calc` is a tactic, as an alias for `apply (exact calc ...)`, where `calc` is the structured proof command described above.

4.6 Dependent Types

Dependent types are the defining feature of the *dependent type theory* family of logics. Although you may not be familiar with the terminology, you are likely to be familiar with the concept in some form or other.

Consider a function `pick` that takes a natural number `n` (i.e., a value from $\mathbb{N}$ `= {0, 1, 2, ...}`) and that returns a natural number between `0` and `n`. Intuitively, `pick n` should have the type `{0, 1, ..., n}` (i.e., the type consisting of all natural numbers `i` $\leq$ `n`). In Lean, this is written `{i : ℕ // i ≤ n}`. This would be the type

of `pick n`. Mathematically inclined readers might want to think of `pick` as an $\mathbb{N}$-indexed family of terms

$$(\texttt{pick n} : \{\texttt{i} : \mathbb{N} \mathbin{//} \texttt{i} \leq \texttt{n}\})_{\texttt{n}:\mathbb{N}}$$

in which the type of each term depends on the index—e.g., `pick 5` $: \{\texttt{i} : \mathbb{N} \mathbin{//} \texttt{i} \leq 5\}$. But what would be the type of `pick` itself? We would like to express that `pick` is a function that takes an argument $\texttt{n} : \mathbb{N}$ and that returns a value of type $\{\texttt{i} : \mathbb{N} \mathbin{//} \texttt{i} \leq \texttt{n}\}$. To capture this, we will write

$$\texttt{pick} : (\texttt{n} : \mathbb{N}) \to \{\texttt{i} : \mathbb{N} \mathbin{//} \texttt{i} \leq \texttt{n}\}$$

This is a dependent type: The type of the result depends on the value of the argument `n`. (The variable name `n` itself is immaterial; we could also write `m` or `x`.)

Unless otherwise specified, a *dependent type* means a type depending on a (non-type) term, as above, with $\texttt{n} : \mathbb{N}$ as the term and $\{\texttt{i} : \mathbb{N} \mathbin{//} \texttt{i} \leq \texttt{n}\}$ as the type that depends on it. But:

- A type may also depend on another type—for example, the type constructor `List`, its η-expanded variant $\texttt{fun}\ \alpha : \texttt{Type} \mapsto \texttt{List}\ \alpha$, or the polymorphic type $\texttt{fun}\ \alpha : \texttt{Type} \mapsto \alpha \to \alpha$ of functions with the same domain and codomain.
- A term may depend on a type—for example, the polymorphic identity function $\texttt{fun}\ \alpha : \texttt{Type} \mapsto \texttt{fun}\ \texttt{x} : \alpha \mapsto \texttt{x}$.
- And of course, a term may also depend on a term—for example, $\texttt{fun n} : \mathbb{N} \mapsto \texttt{n} + 2$.

In summary, there are four cases for $\texttt{fun x} \mapsto \texttt{t}$:

Body (t)		Argument (x)	Description
A term	depending on	a term	Simply typed λ-expression
A type	depending on	a term	Dependent type (in the narrow sense)
A term	depending on	a type	Polymorphic term
A type	depending on	a type	Type constructor

The last three rows correspond to the three axes of Henk Barendregt's λ-cube.[1]

The App and Fun rules presented in Section 1.3 must be generalized to work with dependent types:

$$\frac{\texttt{C} \vdash \texttt{t} : (\texttt{x} : \sigma) \to \tau[\texttt{x}] \qquad \texttt{C} \vdash \texttt{u} : \sigma}{\texttt{C} \vdash \texttt{t u} : \tau[\texttt{u}]}\ \textsc{App}'$$

$$\frac{\texttt{C}, \texttt{x} : \sigma \vdash \texttt{t} : \tau[\texttt{x}]}{\texttt{C} \vdash (\texttt{fun x} : \sigma \mapsto \texttt{t}) : (\texttt{x} : \sigma) \to \tau[\texttt{x}]}\ \textsc{Fun}'$$

The notation $\tau[\texttt{x}]$ stands for a type that may contain `x`, and $\tau[\texttt{u}]$ stands for the same type where all occurrences of `x` have been replaced by `u`.

The simply typed case arises when `x` does not occur in $\tau[\texttt{x}]$. Then, we can simply write $\sigma \to$ instead of $(\texttt{x} : \sigma) \to$. The familiar notation $\sigma \to \tau$ is equivalent

[1] https://en.wikipedia.org/wiki/Lambda_cube

to $(\_ : \sigma) \to \tau$. It is easy to check that App′ and Fun′ coincide with App and Fun when x does not occur in $\tau[x]$.

The example below demonstrates App′:

$$\frac{\vdash \texttt{pick} : (n : \mathbb{N}) \to i : \mathbb{N} \,//\, i \le n \qquad 5 : \mathbb{N}}{\vdash \texttt{pick}\ 5 : i : \mathbb{N} \,//\, i \le 5}\ \textsc{App}'$$

The next example demonstrates Fun′:

$$\frac{\dfrac{\alpha : \texttt{Type}, x : \alpha \vdash x : \alpha}{\alpha : \texttt{Type} \vdash (\texttt{fun}\ x : \alpha \mapsto x) : \alpha \to \alpha}\ \textsc{Fun or Fun}'}{\vdash (\texttt{fun}\ \alpha : \texttt{Type} \mapsto \texttt{fun}\ x : \alpha \mapsto x) : (\alpha : \texttt{Type}) \to \alpha \to \alpha}\ \textsc{Fun}'$$

The picture is incomplete because we only check that the terms—the entities on the left-hand side of a colon (:)—are well typed. The types—the entities on the right-hand side of a colon—should also be checked, using the same type system. For example, the type of `Nat.succ` is $\mathbb{N} \to \mathbb{N}$, whose type is `Type`. Types of types, such as `Type` and `Prop`, are called universes. We will study them more closely in Chapter 12.

Remarkably, universal quantification is simply an alias for a dependent type: $\forall x : \sigma, \tau$ abbreviates $(x : \sigma) \to \tau$. This will become clearer below.

4.7 The PAT Principle

You will likely have noticed that the same symbol $\to$ is used both for implication (e.g., `False` $\to$ `True`) and as the type constructor of functions (e.g., $\mathbb{Z} \to \mathbb{N}$). Without context, we cannot tell whether `a` $\to$ `b` refers to the type of a function with domain `a` and codomain `b` or to the proposition "`a` implies `b`."

It turns out that not only the two concepts *look* the same, they *are* the same. This is called the *PAT principle*, where PAT is a double mnemonic:

PAT = propositions as types PAT = proofs as terms

Furthermore, because types are also terms, we also have that propositions are terms. However, PAT is not a quadruple mnemonic (~~PAT = proofs as types~~). Also note that not all types are propositions, and not all terms are proofs.

By using terms and types to represent proofs and propositions, dependent type theory achieves a considerable economy of concepts. The question "Is `H` a proof of `P`?" becomes equivalent to "Does the term `H` have type `P`?" As a result, inside of Lean, there is no proof checker, only a type checker.

Let us review the dramatis personae one by one. We use the mathematical variables σ, τ for types; `P`, `Q` for propositions; `t`, `u`, `x` for terms; and `h`, `G`, `H`, for proofs.

Starting with "propositions as types," for types, we have the following:

- $\sigma \to \tau$ is the type of functions from σ to τ.
- $(x : \sigma) \to \tau[x]$ is the dependent type of functions from $x : \sigma$ to $\tau[x]$.

In contrast, for propositions, we have the following:

- `P → Q` can be read as "`P` implies `Q`," or as the type of functions mapping proofs of `P` to proofs of `Q`.
- `∀x : σ, Q[x]` can be read as "for all x, `Q[x]`," or as the type of functions of type `(x : σ) → Q[x]`, mapping values `x` of type `σ` to proofs of `Q[x]`.

Continuing with "proofs as terms," for terms, we have the following:

- A constant is a term.
- A variable is a term.
- `t u` is the application of function `t` to argument `u`.
- `fun x ↦ t[x]` is a function mapping `x` to `t[x]`.

In contrast, for proofs (i.e., proof terms), we have the following:

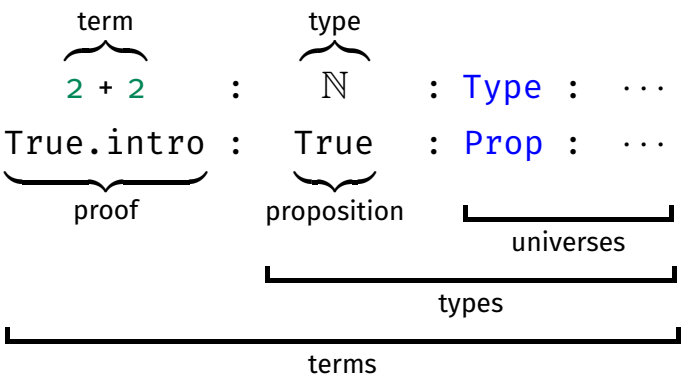
- The name of a theorem or hypothesis is a proof.
- `H t`, which instantiates the leading parameter or `∀` quantifier of proof `H`'s statement with term `t`, is a proof.
- `H G`, which discharges the leading assumption of `H`'s statement with proof `G`, is a proof. This operation is called modus ponens.
- `fun h : P ↦ H[h]` is a proof of `P → Q`, assuming `H[h]` is a proof of `Q` for all `h : P`.
- `fun x : σ ↦ H[x]` is a proof of `∀x : σ, Q[x]`, assuming `H[x]` is a proof of `Q[x]` for all `x : σ`.

The last two cases are justified by the `Fun′` rule. In a structured proof, as opposed to a raw proof term, we would write `assume` or `fix` instead of `fun`, and we would probably want to repeat the conclusion using `show` for readability, as follows:

```
theorem case_4 :
  P → Q :=
assume h : P
show Q from
  H[h]
```

```
theorem case_5 :
  ∀x : σ, Q[x] :=
fix x : τ
show Q[x] from
  H[x]
```

The terminology of dependent type theory can be quite confusing, because some words have a narrow and a broad sense. The following diagram captures the various meanings of important words:



According to the broad senses, any expression is a term, any expression that may occur on the right-hand side of a typing judgment is a type, and any expression

that may occur on the right-hand side of a typing judgment with a type on its left-hand side is a universe. This is consistent with the reading of `t : u` as "`t` has type `u`" and the notion that universes are types of types.

Some commands are provided in Lean under two names but are essentially the same by the PAT principle. This is the case for `fix` and `assume`; indeed, both are defined by `LoVelib` as aliases for `fun`. There are also pairs with slightly different behavior, such as `def`/`theorem` and `let`/`have`. The fundamental difference is this: When we define some function or data, we care not only about the type but also about the body—the behavior. On the other hand, once we have proved a theorem, the proof becomes *irrelevant*. All that matters is that there *is* a proof. We will return to the topic of proof irrelevance in Chapter 12.

The following correspondence table summarizes the differences between tactical proofs, structured proofs, and raw proof terms:

Tactical proof	Structured proof	Raw proof term
`intro x`	`fix x : τ`	`fun x ↦`
`intro h`	`assume h : P`	`fun h ↦`
`have k := H`	`have k := H`	`(fun k ↦ ...) H`
`let x := t`	`let x := t`	`(fun x ↦ ...) t`
`exact (H : P)`	`show P from H`	`H : P`
`calc ...`	`calc ...`	`calc ...`

Notice that `let x := t` followed by `u` is essentially a sophisticated way to write `(fun x ↦ u) t`.

4.8 Induction by Pattern Matching and Recursion

In Section 3.6, we reviewed the use of the `induction` tactic to perform proofs by induction. An alternative, more flexible style relies on pattern matching and the PAT principle.

Recall the definition of the reverse of a list from Section 2.2:

```
def reverse {α : Type} : List α → List α
  | []      => []
  | x :: xs => reverse xs ++ [x]
```

In fact, `reverse` exists as `List.reverse` in Lean's standard library, but our definition is optimized for reasoning. A useful property to prove is that `reverse` is its own inverse: `reverse (reverse xs) = xs` for all lists `xs`. However, if we try to prove it by induction, we quickly run into an obstacle. The induction step is

```
ih : ∀xs, reverse (reverse xs) = xs ⊢ reverse (reverse xs ++ [x]) = x :: xs
```

Notice the unpleasant presence of `++ [x]` inside the double `reverse` sandwich. We need a way to "distribute" the outer `reverse` over `++` to obtain a term that matches the induction hypothesis's left-hand side. The trick is to prove and use the following theorem:

```
theorem reverse_append {α : Type} :
  ∀xs ys : List α,
```

```
    reverse (xs ++ ys) = reverse ys ++ reverse xs
  | [],      ys => by simp [reverse]
  | x :: xs, ys => by simp [reverse, reverse_append xs]
```

The theorem's proof arguably looks more like a recursive function definition than a proof. The patterns on the left, `[]` and `x :: xs`, correspond to the two constructors of the ∀-quantified variable `xs`. On the right-hand side of each `:=` symbol is a proof for the corresponding case. The variables on which we can pattern-match are those that appear in the ∀ quantifiers, in order of occurrence (here, `xs` and `ys`). Inside the induction step's proof, the induction hypothesis is available under the same name as the theorem we are proving (`reverse_append`).

We explicitly pass `xs` as argument to the induction hypothesis. This restricts the hypothesis so that it only applies to `xs` but not to other lists. In particular, this ensures that the theorem will not be applied to `x :: xs`, which could lead to a circular argument: "To prove `reverse_append (x :: xs)`, use `reverse_append (x :: xs)`." Lean's termination checker would notice that the argument is ill-founded and raise an error, but we want to avoid that. In addition, the explicit argument `xs` is useful as documentation, effectively saying, "The only recursive instance of the theorem we need is the one on `xs`."

For reference, a tactical proof would be as follows:

```
theorem reverse_append_tactical {α : Type} (xs ys : List α) :
  reverse (xs ++ ys) = reverse ys ++ reverse xs :=
  by
    induction xs with
    | nil          => simp [reverse]
    | cons x xs' ih => simp [reverse, ih]
```

The theorem would also be provable, and useful, if we put `[y]` instead of `ys`. But it is a good habit to state theorems as generally as possible. This results in more reusable libraries. Moreover, this is often necessary to obtain a strong enough induction hypothesis in a proof by induction. In general, finding the right inductions and theorems can require thought and creativity.

Simultaneous pattern matching on multiple variables is supported (e.g., `xs` and `ys` above). The patterns are then separated by commas. The general format is

```
theorem name (params₁ : type₁) ... (paramsₘ : typeₘ) :
  statement
  | patterns₁ => proof₁
    ⋮
  | patternsₙ => proofₙ
```

Notice the strong similarity with the syntax of `def` (Section 2.2). The two commands are, in fact, almost the same, but `theorem` considers the defined term or the proof opaque, whereas `def` keeps it transparent. Since the actual proofs are irrelevant once a theorem is proved (Section 12), there is no need to expand them later. A similar distinction exists between `let` and `have`.

By the PAT principle, a proof by induction by pattern matching and recursion is the same as a recursive proof term. When we invoke the induction hypothesis, we are really just invoking a recursive function recursively. This explains why the

induction hypothesis has the same name as the theorem we prove. Lean's termination checker is used to establish well-foundedness of the proof by induction.

With the `reverse_append` theorem in place, we can return to our initial goal:

```
theorem reverse_reverse {α : Type} :
  ∀xs : List α, reverse (reverse xs) = xs
  | []      => by rfl
  | x :: xs =>
    by simp [reverse, reverse_append, reverse_reverse xs]
```

Induction by pattern matching and recursion is popular among Lean users. Its main advantages are its convenient syntax and its support for well-founded induction, which is more powerful than structural induction as provided by the `induction` tactic (Section 3.7). However, in this guide, we will not need the full power of well-founded induction. Furthermore, for subtle logical reasons, induction by pattern matching and recursion is not available for inductive predicates, which are the topic of Chapter 6. For these reasons, we will often prefer the `induction` tactic.

4.9 Summary of New Lean Constructs

Proof Commands

`assume`	states assumptions
`calc`	combines proofs by transitivity
`fix`	fixes variables
`have`	states an intermediate theorem
`let`	introduces a local definition
`show`	states the target

Tactics

`calc`	combines proofs by transitivity
`have`	states an intermediate theorem
`let`	introduces a local definition

Part II

Functional–Logic Programming

Chapter 5

Functional Programming

We take a closer look at the essence of typed functional programming: inductive types, proofs by induction, recursive functions, pattern matching, structures (records), and type classes. The concepts covered here are described in more detail in Chapters 7 to 10 of *Theorem Proving in Lean 4* [13].

5.1 Inductive Types

Inductive types are modeled after the data types of typed functional programming languages (e.g., Haskell, ML, OCaml). They are also reminiscent of sealed classes in Scala. Already in Chapter 2, we saw some basic inductive types: the natural numbers, the finite lists, and a type of arithmetic expressions. In this chapter, we revisit the lists and study binary trees. We also take a brief look at vectors of length *n*, a dependent type.

Recall the definition of natural numbers as an inductive type:

```
inductive Nat : Type where
  | zero : Nat
  | succ : Nat → Nat
```

This definition introduces the type `Nat` and the two constants `Nat.zero` and `Nat.succ`, called constructors. The definition also asserts some properties of the constructors, which is why we use the `inductive` command. In addition, it introduces further constants that are used internally to support induction and recursion.

As we saw in Section 2.1, an inductive type is a type whose members are all the values that can be built by a finite number of applications of its constructors, and only those. Mottos:

- *No junk*: The type contains no values beyond those expressible using the constructors.
- *No confusion*: Values built using a different combination of constructors are different.

For natural numbers, "no junk" means that there exist no special values such as -1, ε, ∞, or `NaN` that cannot be expressed using a finite combination of `Nat.zero` and `Nat.succ`, and "no confusion" ensures that `Nat.zero` $\neq$ `Nat.succ n` for all `n` and that `Nat.succ` is injective. In addition, values of inductive types are always

finite. The infinite term

```
Nat.succ (Nat.succ (Nat.succ (Nat.succ ...)))
```

is not a value. Nor does there exist a value `n` such that `Nat.succ n = n`, as we will show below.

Inductive types are very convenient to use, because they support induction and recursion and their constructors are well behaved, but not all types can be defined as an inductive type. In particular, mathematical types such as $\mathbb{Q}$ (the rationals) and $\mathbb{R}$ (the real numbers) require more elaborate constructions, based on quotienting and subtyping. This will be explained in Chapters 12 and 14.

5.2 Structural Induction

Structural induction is a generalization of mathematical induction to arbitrary inductive types. To prove a goal `n : ℕ ⊢ P[n]` by structural induction on `n`, it suffices to show two subgoals, traditionally called the base case and the induction step:

```
⊢ P[0]
k : ℕ, ih : P[k] ⊢ P[k + 1]
```

We can of course also write `P[Nat.zero]` and `P[Nat.succ k]`.

In general, the situation is more complex. The goal might contain some extra hypotheses (e.g., `Q`) that do not depend on `n` and others (e.g., `R[n]`) that do. Assuming we have one hypothesis of each kind, this gives the initial goal

```
hQ : Q, n : ℕ, hR : R[n] ⊢ S[n]
```

Structural induction on `n` then produces the two subgoals

```
hQ : Q, hR : R[0] ⊢ S[0]
hQ : Q, k : ℕ, ih : R[k] → S[k], hR : R[k + 1] ⊢ S[k + 1]
```

The hypothesis `Q` is simply carried over unchanged from the initial goal, whereas `R[n] ⊢ S[n]` is treated almost the same as if the goal's target had been `R[n] → S[n]`. This is easy to check by taking `P[n] := R[n] → S[n]` in the first example above. Since this general format is very verbose and hardly informative (now that we understand how it works), from now on we will present goals in the simplest form possible, without extra hypotheses.

For lists, given a goal `xs : list α ⊢ P[xs]`, structural induction on `xs` yields

```
⊢ P[[]]
y : α, ys : list α, ih : P[ys] ⊢ P[y :: ys]
```

We can of course also write `List.nil` and `List.cons y ys`. There is no induction hypothesis associated with `y`, because `y` is not of list type.

For arithmetic expressions, the base cases are

```
i : ℤ ⊢ P[AExp.num i]        x : String ⊢ P[AExp.var x]
```

and the induction steps are

```
e₁ e₂ : AExp, ih₁ : P[e₁], ih₂ : P[e₂] ⊢ P[AExp.add e₁ e₂]
e₁ e₂ : AExp, ih₁ : P[e₁], ih₂ : P[e₂] ⊢ P[AExp.sub e₁ e₂]
e₁ e₂ : AExp, ih₁ : P[e₁], ih₂ : P[e₂] ⊢ P[AExp.mul e₁ e₂]
e₁ e₂ : AExp, ih₁ : P[e₁], ih₂ : P[e₂] ⊢ P[AExp.div e₁ e₂]
```

Notice the two induction hypotheses, about `e₁` and `e₂`.

In general, structural induction produces one subgoal per constructor. In each subgoal, induction hypotheses are available for all constructor arguments of the type we are performing the induction on.

Given an inductive type τ, the procedure to compute the subgoals is always the same:

1. Replace the hole in `P[ ]` with each possible constructor applied to fresh variables (e.g., `y :: ys`), yielding as many subgoals as there are constructors.
2. Add these new variables (e.g., `y`, `ys`) to the local context.
3. Add induction hypotheses for all new variables of type τ.

As an example, we will prove that `Nat.succ n` $\neq$ `n` for all `n` : $\mathbb{N}$. We start with an informal proof:

> The proof is by structural induction on `n`.
>
> Case `0`: We must show `Nat.succ 0` $\neq$ `0`. This follows from the "no confusion" property of the constructors of inductive types.
>
> Case `Nat.succ k`: The induction hypothesis is `Nat.succ k` $\neq$ `k`. We must show `Nat.succ (Nat.succ k)` $\neq$ `Nat.succ k`. By the injectivity of `Nat.succ`, we have that `Nat.succ (Nat.succ k) = Nat.succ k` is equivalent to `Nat.succ k = k`. Thus, it suffices to prove `Nat.succ k` $\neq$ `k`, which corresponds exactly to the induction hypothesis. □

Notice the main features of this informal proof, which you should aim to reproduce in your own informal arguments:

- The proof starts with an unambiguous announcement of the type of proof we are carrying out (e.g., which kind of induction and on which variable).
- The cases are clearly identified, and for each case, both the goal's target and the hypotheses are stated.
- The key theorems on which the proof relies are explicitly invoked (e.g., injectivity of `Nat.succ`).

Now let us carry out the proof in Lean:

```
theorem Nat.succ_neq_self (n : ℕ) :
  Nat.succ n ≠ n :=
by
  induction n with
  | zero       => simp
  | succ n' ih => simp [ih]
```

The routine reasoning about constructors is all carried out by `simp`.

5.3 Structural Recursion

Structural recursion is a form of recursion that allows us to peel off one constructor from the value on which we recurse. The factorial function below is structurally recursive:

```
def fact : ℕ → ℕ
  | 0     => 0
  | n + 1 => (n + 1) * fact n
```

The constructor we peel off here is `Nat.succ` (written `+ 1`). Such functions are guaranteed to call themselves only finitely many times before the recursion stops; for example, `fact 12345` will call itself 12345 times. The function is said to *terminate*. This property helps ensure logical consistency.

With structural recursion, there are as many equations as there are constructors. Novices are often tempted to supply additional, redundant cases, as in the following example:

```
def factThreeCases : ℕ → ℕ
  | 0     => 0
  | 1     => 1
  | n + 1 => (n + 1) * factThreeCases n
```

It is in your own best interest to resist this temptation. The more cases you have in your definitions, the more work it will be to reason about them. Keep in mind the saying that one good definition is worth three theorems.

For structurally recursive functions, Lean can automatically prove termination. For more general recursive schemes, the termination check may fail. Sometimes it does so for a good reason, as in the following example:

```
-- fails
def illegal : ℕ → ℕ
  | n => illegal n + 1
```

If Lean were to accept this definition, we could exploit it to prove that `0 = 1`, by subtracting `illegal n` from each side of the equation `illegal n = illegal n + 1`. From `0 = 1`, we could derive `False`, and from `False`, we could derive anything. Clearly, we do not want that.

If we had used `opaque` and `axiom`, nothing could have saved us:

```
opaque immoral : ℕ → ℕ

axiom immoral_eq (n : ℕ) :
  immoral n = immoral n + 1

theorem proof_of_False :
  False :=
  have hi : immoral 0 = immoral 0 + 1 :=
    immoral_eq 0
  have him :
    immoral 0 - immoral 0 = immoral 0 + 1 - immoral 0 :=
    by rw [←hi]
  have h0eq1 : 0 = 1 :=
```

```
    by simp at him
  show False from
    by simp at h0eq1
```

Another reason for preferring `def` over `opaque` and `axiom` is that the defining equations can be used in computations. Tactics such as `rfl` that unify up to computation become stronger each time we introduce a definition, and the diagnosis commands `#eval` and `#reduce` can be used on defined constants.

The observant reader will have noticed that the above definitions of factorial are mathematically wrong: `fact` shockingly returns `0` regardless of the argument. We quite literally `fact`ed up. These embarrassing mistakes remind us to *test* our definitions and *prove* some of their properties. Although flawed axioms arise now and then, what is much more common are definitions that fail to capture the intended concepts. Just because a function is called `fact` does not mean that it actually computes factorials.

5.4 Pattern Matching Expressions

Pattern matching is possible not only at the top level of a `def` command but also deeply within terms, via a `match` expression. The construct has the following general syntax:

```
match term₁, ..., termₘ with
  | pattern₁₁, ..., pattern₁ₘ => result₁
      ⋮
  | patternₙ₁, ..., patternₙₘ => resultₙ
```

The construct is vaguely reminiscent of `switch` in many programming languages. An English translation of the above `match` expression follows:

> Consider the terms $term_1, \ldots, term_m$.
>
> If they are respectively of the forms $pattern_{11}, \ldots, pattern_{1m}$, then yield $result_1$.
>
> ⋮
>
> If they are respectively of the forms $pattern_{n1}, \ldots, pattern_{nm}$, then yield $result_n$.

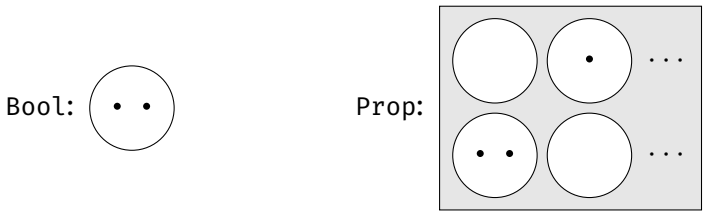
The patterns may contain variables, constructors, and nameless placeholders (`_`). The $result_i$ expressions may refer to the variables introduced in the corresponding patterns.

The following function definition demonstrates the syntax of pattern matching within expressions:

```
def bcount {α : Type} (p : α → Bool) : List α → ℕ
  | []      => 0
  | x :: xs =>
    match p x with
    | true  => bcount p xs + 1
    | false => bcount p xs
```

The `bcount` function counts the number of elements in a list that satisfy the given predicate `p`. The predicate's codomain is `Bool`. As a general rule, we will use type `Bool`, of Booleans, within programs and use the type `Prop`, of propositions, when stating properties of programs. The two values of type `Bool` are called `false` and `true` (in lowercase).[1] The connectives are called `or` (infix: `||`), `and` (infix: `&&`), and `not` (prefix: `!`).

The following diagram shows the interpretations of `Bool` and `Prop`:



Dots represents elements, circles represents types, and the rectangle represents a type of types, or universe. We see that `Bool` is interpreted as a set with two values, whereas `Prop` consists in an infinite number of propositions (the types), each of which has zero or more proofs (the elements). We will refine this picture in Chapter 6.

We cannot `match` on a proposition (of type `Prop`), but we can use `if–then–else` instead. For example, the `min` operator on natural numbers operator can be defined as follows:

```
def min (a b : ℕ) : ℕ :=
  if a ≤ b then a else b
```

This requires a decidable (i.e., executable) proposition. This is the case for $\leq$: Given concrete values for the arguments, such as 35 and 49, Lean can reduce $35 \leq 49$ to `True`. Lean keeps track of decidability using a mechanism called type classes, which will be explained below.

5.5 Structures

Lean provides a convenient syntax for defining records, or *structures* as they are also called. These are essentially nonrecursive single-constructor inductive types but with some syntactic sugar.

The definition below introduces a structure called `RGB` with three fields of type `ℕ` called `red`, `green`, and `blue`:

```
structure RGB where
  red   : ℕ
  green : ℕ
  blue  : ℕ
```

This definition has roughly the same effect as the following commands:

```
inductive RGB : Type where
  | mk : ℕ → ℕ → ℕ → RGB
```

[1] Lean also allows us to use `True` and `False`, but these are then implicitly converted from `Prop` to `Bool`. We generally recommend avoiding such implicit coercions. There is unfortunately no way to disable them.

```
def RGB.red : RGB → ℕ
  | RGB.mk r _ _ => r

def RGB.green : RGB → ℕ
  | RGB.mk _ g _ => g

def RGB.blue : RGB → ℕ
  | RGB.mk _ _ b => b
```

We can define a new structure as the extension of an existing structure. The definition below extends `RGB` with a fourth field, called `alpha`:

```
structure RGBA extends RGB where
  alpha : ℕ
```

The general syntax to define structures is

```
structure structure-name (params₁ : type₁) ... (paramsₖ : typeₖ)
  [extends structure₁, ..., structureₘ] where
  field-name₁ : field-type₁
    ⋮
  field-nameₙ : field-typeₙ
```

The parameters $params_1$, ..., $params_k$ are effectively additional fields, but unlike $field\text{-}name_1$, ..., $field\text{-}name_n$, they are stored in the type, as arguments to the type constructor (*structure-name*).

Values can be specified in a variety of syntaxes:

```
def pureRed : RGB :=
  RGB.mk 0xff 0x00 0x00

def pureGreen : RGB :=
  { red   := 0x00
    green := 0xff
    blue  := 0x00 }

def semitransparentGreen : RGBA :=
  { pureGreen with
    alpha := 0x7f }
```

The definition of `semitransparentGreen` copies all the values from `pureGreen` except for the `alpha` field, which it sets explicitly.

Next, we define an operation called `shuffle`:

```
def shuffle (c : RGB) : RGB :=
  { red   := RGB.green c
    green := RGB.blue c
    blue  := RGB.red c }
```

The definition relies on the generated selectors `RGB.red`, `RGB.green`, and `RGB.blue`. Instead of `RGB.red c`, we could also write `c.red`, and similarly for the other fields. Sometimes we will see this notation in Lean's output, even if we do not use it ourselves.

Applying `shuffle` three times in a row is the same as not applying it at all:

```
theorem shuffle_shuffle_shuffle (c : RGB) :
  shuffle (shuffle (shuffle c)) = c :=
  by rfl
```

5.6 Type Classes

Type classes are a mechanism that was popularized by Haskell and that is present in several proof assistants. In Lean, a *type class* is a structure type combining abstract constants and their properties.[2] A type can be declared an instance of a type class by providing concrete definitions for the constants and proving that the properties hold. Based on the type, Lean retrieves the relevant instance.

A simple example is the type class `Inhabited`, which requires only a constant `Inhabited.default` and no properties:

```
class Inhabited (α : Type) : Type where
  default : α
```

The syntax is the same as for structures, except with the keyword `class` instead of `structure`. The parameter α represents an arbitrary type that could be a member of this class. This particular type class has a single parameter and a single field, but in general a type class can have multiple parameters and multiple fields.

Any type that has at least one element can be registered as an instance of the `Inhabited` type class. For example, we can register $\mathbb{N}$ by choosing an arbitrary number to be the default value:

```
instance Nat.Inhabited : Inhabited ℕ :=
  { default := 0 }
```

This specifies a value called `Nat.Inhabited` of type `Inhabited ℕ`. Because we use the keyword `instance` instead of `def`, the structure value is registered as *the* canonical instance to use whenever a structure of type `Inhabited ℕ` is desired. In the global table storing type class instances, there is now an entry

$$\texttt{Inhabited}\ \mathbb{N} \mapsto \texttt{Nat.Inhabited}$$

For lists, the empty list is an obvious default value that can be constructed even if α is not inhabited:

```
instance List.Inhabited {α : Type} : Inhabited (List α) :=
  { default := [] }
```

This adds the following entry to the global table:

$$\texttt{Inhabited (List ?}\alpha\texttt{)} \mapsto \texttt{List.Inhabited}$$

Sometimes we may want to supply several instances of a given type class for the same type. Lean will then choose the first matching instance it finds.

As an example, observe that finite functions of type $\alpha \to \beta$ can be represented by their function tables, of type $\beta \times \cdots \times \beta$ (with $|\alpha|$ copies of β). Accordingly,

[2] Despite its name, Lean's type class mechanism is more closely related to Scala's implicit arguments than to Haskell's type classes.

$|\alpha \to \beta| = |\beta|^{|\alpha|}$. To make this 0, we must have both $|\beta| = 0$ and $|\alpha| \neq 0$. In other words, the type $\alpha \to \beta$ is inhabited if either (1) β is inhabited or (2) α is *not* inhabited. We focus on case (1):

```
instance Fun.Inhabited {α β : Type} [Inhabited β] :
  Inhabited (α → β) :=
  { default := fun a : α ↦ Inhabited.default }
```

The instance relies itself on an instance of the same type class but on a different type. This occurs frequently.

The type $\alpha \times \beta$ of pairs, also called product type, contains values of the form `(a, b)`, where `a : α` and `b : β`. Given a pair `ab : α × β`, the first and second components can be extracted by writing `Prod.fst ab` and `Prod.snd ab`. To provide an inhabitant of $\alpha \times \beta$, we need both an inhabitant of α and an inhabitant of β:

```
instance Prod.Inhabited {α β : Type}
    [Inhabited α] [Inhabited β] :
  Inhabited (α × β) :=
  { default := (Inhabited.default, Inhabited.default) }
```

Using the `Inhabited` type class, we can define the head operation on lists: the function that returns the first element of a list. Because an empty list contains no elements, there is no meaningful value we can return in that case. Given a type that belongs to the `Inhabited` type class, we can simply return the default value:

```
def head {α : Type} [Inhabited α] : List α → α
  | []     => Inhabited.default
  | x :: _ => x
```

We require that α belongs to `Inhabited` by writing `[Inhabited α]`. This allows us to access `Inhabited.default` in the definition.

The syntax `[Inhabited α]` adds an implicit argument of the `head` constant. But unlike for other implicit arguments, Lean performs a type class search through all declared instances to determine the value of this argument. Thus, when running the command

```
#eval head ([] : List ℕ)
```

Lean will look for an `Inhabited` $\mathbb{N}$ instance and find `Nat.Inhabited`, the instance we declared above. In that declaration, we set `default` to be 0 and hence this is what `#eval` prints. If multiple instances are applicable and Lean chooses the wrong one, we can use the `@` syntax to transform type class arguments into explicit arguments and supply the desired type class instance.

Let us take a closer look at `Inhabited.default`:

```
Inhabited.default {α : Type} [Inhabited α] : α
```

Notice that the selectors of plain structures use parentheses `( )` whereas those of type classes use square brackets `[ ]`. When we used `Inhabited.default α` to define `head`, Lean looked for an instance of `Inhabited α` in the global table of registered instances and in the local context. The global table contained an entry for `Inhabited` $\mathbb{N}$ but none that matched `Inhabited α`. On the other hand, the local context contained an anonymous parameter of type `Inhabited α`, which could be used.

Lean's core library defines `List.head` exactly as we did. In practice, almost all types are nonempty (with the notable exception of `False`), so the `Inhabited` restriction is hardly an issue.

We can prove abstract theorems about the `Inhabited` type class, such as

```
theorem head_head {α : Type} [Inhabited α] (xs : List α) :
  head [head xs] = head xs
```

The assumption `[Inhabited α]` is needed to use the operator `head` on lists of type `List α`. If we omit this assumption, Lean will raise an error telling us that type class synthesis failed.

There are more type classes with only a constant but no properties, including

```
class Zero (α : Type) where
  zero : α

class One (α : Type) where
  one : α

class Neg (α : Type) where
  neg : α → α

class Inv (α : Type) where
  inv : α → α

class Add (α : Type) where
  add : α → α → α

class Mul (α : Type) where
  mul : α → α → α
```

These *syntactic* type classes introduce constants that are used in different contexts with different semantics. For example, `one` can stand for the natural number 1, the integer 1, the real 1, the identity matrix, and many other concepts of 1. The main purpose of these type classes is to form the foundation for a rich hierarchy of algebraic type classes (groups, monoids, ring, fields, etc.) and to allow the overloading of common mathematical symbols such as `+`, `*`, `0`, `1`, and $^{-1}$.

Syntactic type classes do not impose severe restrictions on the types that can be declared instances. In contrast, the *semantic* type classes contain properties that restrict how the given constants behave.

In Section 3.6, we encountered the following semantic type classes:

```
class IsCommutative (α : Type) (f : α → α → α) where
  comm : ∀a b, f a b = f b a

class IsAssociative (α : Type) (f : α → α → α) where
  assoc : ∀a b c, f (f a b) c = f a (f b c)
```

This time, the associations are not from a type to a constant, but from a type *and a function* to a property. Lean does not mind the abuse: Although they are called type classes, Lean's type classes are very flexible and can be used to express all sorts of constraints.

Conceptually, `IsCommutative` is a dependent type of triples `(α, f, comm)`, and similarly for `IsAssociative`. The type of `f` depends on `α`, and the type of `comm` depends on `α` and `f`. Although they are parameters, `α` and `f` are also stored along with `comm`.

In Section 3.6, we registered our `add` function on $\mathbb{N}$ as a commutative and associative operation:

```
instance IsAssociative_add : IsAssociative ℕ add :=
  { assoc := add_assoc }
```

```
instance IsCommutative_add : IsCommutative ℕ add :=
  { comm := add_comm }
```

Whenever we try to access `@IsCommutative.comm ℕ add`, we obtain `add_comm`, and similarly for `@IsAssociative.assoc ℕ add`. The `ac_rfl` tactic tries to look up the `comm` and `assoc` properties for all binary operators in the problem and exploits the properties whenever they are present.

The general syntax to define a type class is as follows:

```
class class-name (params₁ : type₁) ... (paramsₖ : typeₖ)
  [extends structure₁, ..., structureₘ] where
  constant-name₁ : constant-type₁
    ⋮
  constant-nameₙ : constant-typeₙ
  property-name₁ : proposition₁
    ⋮
  property-nameₚ : propositionₚ
```

The general syntax to instantiate a type class is as follows:

```
instance instance-name : type-class arguments :=
  { constant₁ := definition₁,
      ⋮
    constantₙ := definitionₙ,
    property₁ := proof₁,
      ⋮
    propertyₚ := proofₚ }
```

5.7 Lists

Lean provides a rich library of functions on finite lists. In this section, we will review some of them, and we will define some of our own; these are good exercises to familiarize ourselves with functional programming in Lean.

In the first example, we perform a case distinction on a list:

```
theorem head_head_cases {α : Type} [Inhabited α]
    (xs : List α) :
  head [head xs] = head xs :=
  by
    cases xs with
    | nil        => rfl
    | cons x xs' => rfl
```

The proof relies on `cases`, a relative of `induction`. It performs a case distinction on its argument but does not generate induction hypotheses. The invocation `cases xs` transforms a goal `⊢ P[xs]` into two subgoals, `⊢ P[[]]` and `⊢ P[x :: xs']`. We could also have used `induction xs`. If you find yourself hesitating between `induction` and `cases`, you can always choose `induction` and see afterwards if

you needed the induction hypotheses—if you did not need it, it is good style to document that fact by replacing `induction` by `cases`.

In a structured proof, we can use `match` expressions to perform a case distinction:

```
theorem head_head_match {α : Type} [Inhabited α]
    (xs : List α) :
  head [head xs] = head xs :=
  match xs with
  | List.nil        => by rfl
  | List.cons x xs' => by rfl
```

In the next example, we show how to exploit injectivity of constructors. The `cases` tactic can be used to exploit injectivity to simplify equations in which both sides have the same constructor applied. In the proof below, the equation before simplification is `x :: xs = y :: ys`:

```
theorem injection_example {α : Type} (x y : α) (xs ys : List α)
    (h : x :: xs = y :: ys) :
  x = y ∧ xs = ys :=
  by
    cases h
    simp
```

The `cases` tactic replaces `y` by `x` and `ys` by `xs` throughout the goal, yielding the subgoal

```
⊢ x = x ∧ xs = xs
```

which `simp` can easily prove.

The `cases` tactic is also useful when the constructors are different, to detect the impossible case:

```
theorem distinctness_example {α : Type} (y : α) (ys : List α)
    (h : [] = y :: ys) :
  false :=
  by cases h
```

Next, we define the *map function* on lists: a function that applies its argument `f`—which is itself a function—to all elements stored in a container.

```
def map {α β : Type} (f : α → β) : List α → List β
  | []      => []
  | x :: xs => f x :: map f xs
```

Notice that because `f` does not change in the recursive call, we put it as a parameter of the entire definition. The alternative, which is the only option for arguments that change in recursive calls, would be as follows:

```
def mapArgs {α β : Type} : (α → β) → List α → List β
  | _, []      => []
  | f, x :: xs => f x :: mapArgs f xs
```

A basic property of map functions is that they have no effect if their argument is the identity function (`fun x ↦ x`):

```
theorem map_ident {α : Type} (xs : List α) :
  map (fun x ↦ x) xs = xs :=
  by
    induction xs with
    | nil           => rfl
    | cons x xs' ih => simp [map, ih]
```

Another basic property is that successive maps can be compressed into a single map, whose argument is the composition of the functions involved:

```
theorem map_comp {α β γ : Type} (f : α → β) (g : β → γ)
    (xs : List α) :
  map g (map f xs) = map (fun x ↦ g (f x)) xs :=
  by
    induction xs with
    | nil           => rfl
    | cons x xs' ih => simp [map, ih]
```

When introducing new operations, it is useful to show how these behave when used in combination with other operations. Here is an example:

```
theorem map_append {α β : Type} (f : α → β)
    (xs ys : List α) :
  map f (xs ++ ys) = map f xs ++ map f ys :=
  by
    induction xs with
    | nil           => rfl
    | cons x xs' ih => simp [map, ih]
```

Remarkably, the last three proofs are textually identical. These are typical `induction-rfl-simp` proofs.

The next list operation removes the first element of a list, returning the tail:

```
def tail {α : Type} : List α → List α
  | []      => []
  | _ :: xs => xs
```

For `[]`, we simply return `[]` as its own tail.

The counterpart of `tail` is a function that extracts the first element of a list. We already reviewed one solution in Section 5.6, using the `Inhabited` type class. Another possible definition uses an `Option` wrapper:

```
def headOpt {α : Type} : List α → Option α
  | []     => Option.none
  | x :: _ => Option.some x
```

The type `Option α` has two constructors: `Option.none` and `Option.some a`, where `a : α`. We use `Option.none` when we have no meaningful value to return and `Option.some` otherwise. We can think of `Option.none` as the null pointer of functional programming, but unlike null pointers (and null references), the type system guards against unsafe dereferences. To retrieve the value stored in an `Option`, we must pattern-match. Schematically:

```
match headOpt xs with
  | Option.none   => handleTheError
  | Option.some x => doSomethingWithValue x
```

We cannot simply write `doSomethingWithValue (headOpt xs)`, because this would be type-incorrect. The type system forces us to think about error handling.

Using the power of dependent types, another way to implement a partial function is to specify a precondition. The caller must then pass a proof that the precondition is satisfied as argument:

```
def headPre {α : Type} : (xs : List α) → xs ≠ [] → α
  | [],     hxs => by simp at *
  | x :: _, hxs => x
```

The `headPre` function takes two explicit arguments. The first argument, `xs`, is a list. The second argument, `hxs`, is a proof of `xs ≠ []`. Since the type `xs ≠ []` of the second argument depends on the first argument, we must use the dependent type syntax `(xs : List α) →` rather than `List α →` so that we can name the first argument. The result of the function is a value of type `α`; thanks to the precondition, there is no need for an `Option` wrapper.

The second argument is used to rule out the case where `xs` is `[]`. In that case, the argument (called `hxs`) is a proof of `[] ≠ []`, which is impossible. The proof derives a contradiction and exploits it to derive an arbitrary `α`. From a contradiction, we can derive anything, even an inhabitant of `α`.

We can then invoke the function as follows:

```
#eval headPre [3, 1, 4] (by simp)
```

This prints `3`. The second argument, `by simp`, is a proof that `[3, 1, 4]` is not `[]`.

Let us move on. Given two lists $[x_1, \ldots, x_n]$ and $[y_1, \ldots, y_n]$ of the same length, the zip operation constructs a list of pairs $[(x_1, y_1), \ldots, (x_n, y_n)]$:

```
def zip {α β : Type} : List α → List β → List (α × β)
  | x :: xs, y :: ys => (x, y) :: zip xs ys
  | [],      _       => []
  | _ :: _,  []      => []
```

The function is also defined if one list is shorter than the other. For example, `zip [a, b, c] [x, y] = [(a, x), (b, y)]`. Notice that the recursion, with three cases, deviates slightly from the structural recursion schema.

The length of a list is defined by recursion:

```
def length {α : Type} : List α → ℕ
  | []      => 0
  | x :: xs => length xs + 1
```

We can say something interesting about the length of `zip`'s result—namely, it is the minimum of the lengths of the two input lists:

```
theorem length_zip {α β : Type} (xs : List α) (ys : List β) :
  length (zip xs ys) = min (length xs) (length ys) :=
  by
    induction xs generalizing ys with
    | nil           => simp [min, length]
    | cons x xs' ih =>
      cases ys with
      | nil        => rfl
      | cons y ys' => simp [zip, length, ih ys', min_add_add]
```

The proof above teaches us yet another trick. The induction hypothesis is

```
ih : ∀ys : list β, length (zip xs ys) = min (length xs) (length ys)
```

Why is there a ∀ quantifier? The `induction xs generalizing ys` tactic generalized the theorem statement so that the induction hypothesis is not restricted to some fixed `ys` as the proof goal but can be used for arbitrary values of `ys`. Such flexibility is needed here because we want to instantiate the quantifier with `ys`'s tail (called `ys'`) and not with `ys` itself.

The proof relies on a theorem about the `min` function that we need to prove ourselves:

```
theorem min_add_add (l m n : ℕ) :
  min (m + l) (n + l) = min m n + l :=
  by
    cases Classical.em (m ≤ n) with
    | inl h => simp [min, h]
    | inr h => simp [min, h]
```

Recall the definition `min a b = (if a ≤ b then a else b)`. To reason about `min`, we often need to perform a case distinction on the condition `a ≤ b`. This is achieved using `cases Classical.em (a ≤ b)`. This creates two subgoals: one with `a ≤ b` as a hypothesis and one with `¬ a ≤ b`.

Here are two different ways to perform a case distinction on a proposition in a structured proof:

```
theorem min_add_add_match (l m n : ℕ) :
  min (m + l) (n + l) = min m n + l :=
  match Classical.em (m ≤ n) with
  | Or.inl h => by simp [min, h]
  | Or.inr h => by simp [min, h]

theorem min_add_add_if (l m n : ℕ) :
  min (m + l) (n + l) = min m n + l :=
  if h : m ≤ n then
    by simp [min, h]
  else
    by simp [min, h]
```

We see again that the mechanisms that are available to write functional programs, such as `match` and `if–then–else`, are also available for writing structured proofs (which are, after all, terms). We can now add a few rows to the table presented at the end of Section 4.7:

Tactical proof	Structured proof	Raw proof term
`cases t`	`match t with ...`	`match t with ...`
`cases Classical.em Q`	`if Q then ... else ...`	`if Q then ... else ...`

We conclude with a distributivity law about `map` and `zip`, expressed using the `Prod.fst` and `Prod.snd` selectors on pairs:

```
theorem map_zip {α α' β β' : Type} (f : α → α') (g : β → β') :
  ∀xs ys,
    map (fun ab : α × β ↦ (f (Prod.fst ab), g (Prod.snd ab)))
      (zip xs ys) =
    zip (map f xs) (map g ys)
  | x :: xs, y :: ys => by simp [zip, map, map_zip f g xs ys]
  | [],      _       => by rfl
  | _ :: _,  []      => by rfl
```

The patterns on the left correspond exactly to the patterns used in the definition of `zip`. This is simpler than performing the induction on `xs` and the case distinction on `ys` separately, as we did when we proved `length_zip`. Good proofs often follow the structure of the definitions they are based on.

In the definition of `zip` and in the proof of `map_zip`, we were careful to specify three nonoverlapping patterns. It is also possible to write equations with overlapping patterns, as in

```
def f {α : Type} : List α → ···
  | [] => ...
  | xs => ... xs ...
```

Since the patterns are applied sequentially, the above command defines the same function as

```
def f {α : Type} : List α → ···
  | []      => ...
  | x :: xs => ... (x :: xs) ...
```

We generally recommend the latter, more explicit style, because it leads to fewer surprises.

5.8 Binary Trees

Inductive types with constructors taking several recursive arguments define tree-like objects. *Binary trees* have nodes with at most two children. A possible definition of binary trees follows:

```
inductive BTree (α : Type) : Type where
  | empty : BTree α
  | node  : α → BTree α → BTree α → BTree α
```

With binary trees, structural induction produces two induction hypotheses, one for each suBTree of an inner node. To prove a goal `t : BTree α ⊢ P[t]` by structural induction on `t`, we need to show the subgoals

```
⊢ P[BTree.empty]
a : α, l r : BTree α, ih_l : P[l], ih_r : P[r] ⊢ P[BTree.node a l r]
```

The tree counterpart to list reversal is the mirror operation:

```
def mirror {α : Type} : BTree α → BTree α
  | BTree.empty      => BTree.empty
  | BTree.node a l r => BTree.node a (mirror r) (mirror l)
```

Mirroring can be defined directly, without appealing to some append operation. As a result, reasoning about `mirror` is simpler than reasoning about `reverse`, as we can see below:

```
theorem mirror_mirror {α : Type} (t : BTree α) :
  mirror (mirror t) = t :=
  by
    induction t with
    | empty                 => rfl
    | node a l r ih_l ih_r => simp [mirror, ih_l, ih_r]
```

A more detailed informal proof would be as follows:

The proof is by structural induction on `t`.

Case `BTree.empty`: We must show that `mirror (mirror BTree.empty) = BTree.empty`. This follows directly from the definition of `mirror`.

Case `BTree.node a l r`: The induction hypotheses are

(`ih_l`) `mirror (mirror l) = l` (`ih_r`) `mirror (mirror r) = r`

We must show `mirror (mirror (BTree.node a l r)) = BTree.node a l r`. We have

```
  mirror (mirror (BTree.node a l r))
= mirror (BTree.node a (mirror r) (mirror l))   (by def. of mirror)
= BTree.node a (mirror (mirror l)) (mirror (mirror r))     (ditto)
= BTree.node a l (mirror (mirror r))                    (by ih_l)
= BTree.node a l r                                      (by ih_r)
```

□

To achieve the same level of detail in the Lean proof, we could use a calculational block (Section 4.4) instead of `simp`:

```
theorem mirror_mirror_calc {α : Type} :
  ∀t : BTree α, mirror (mirror t) = t
  | BTree.empty      => by rfl
  | BTree.node a l r =>
    calc
      mirror (mirror (BTree.node a l r))
      = mirror (BTree.node a (mirror r) (mirror l)) :=
        by rfl
      _ = BTree.node a (mirror (mirror l))
        (mirror (mirror r)) :=
        by rfl
      _ = BTree.node a l (mirror (mirror r)) :=
        by rw [mirror_mirror_calc l]
      _ = BTree.node a l r :=
        by rw [mirror_mirror_calc r]
```

The following theorem will be useful in Chapter 6:

```
theorem mirror_Eq_empty_Iff {α : Type} :
  ∀t : BTree α, mirror t = BTree.empty ↔ t = BTree.empty
  | BTree.empty      => by simp [mirror]
  | BTree.node _ _ _ => by simp [mirror]
```

5.9 Cases Tactic

cases

```
cases term [with
| constructor₁ names₁ => tactics₁
    ⋮
| constructorₙ namesₙ => tacticsₙ]
```

The `cases` tactic performs a case distinction on the specified term. This gives rise to as many subgoals as there are constructors in the definition of the term's type. The tactic is similar to `induction` except that it does not produce induction hypotheses and it automatically excludes impossible cases. The optional names $names_1, \ldots, names_n$ are used for any emerging variables or hypotheses.

```
cases hypothesis-of-the-form-l-equals-r
```

The `cases` tactic can also be used on a hypothesis `h` of the form `l = r`. It matches `r` against `l` and replaces all occurrences of the variables occurring in `r` with the corresponding terms in `l` everywhere in the goal. The remaining hypothesis `l = l` can be removed using `clear h` if desired.

```
cases Classical.em (proposition) with
| inl name-if-true  => tactics-if-true
| inr name-if-false => tactics-if-false
```

The `cases` tactic can also be used to perform a case distinction on a proposition. Two cases emerge: one in which the proposition is true and one in which it is false. The optional names *name-if-true* and *name-if-false* are used for the hypothesis in the true and false cases, respectively.

5.10 Dependent Inductive Types

The inductive types `List α` and `BTree α` fall within the simply typed fragment of Lean. Inductive types may also depend on (non-type) terms. A typical example is the type of lists of length *n*, or *vectors*:

```
inductive Vec (α : Type) : ℕ → Type where
  | nil                                : Vec α 0
  | cons (a : α) {n : ℕ} (v : Vec α n) : Vec α (n + 1)
```

Thus, the term `Vec.cons 3 (Vec.cons 1 Vec.nil)` has type `Vec ℕ 2`. By encoding the vector length in the type, we can provide more precise information about the result of functions. A function such as `Vec.reverse`, which reverses a vector, would map a value `Vec α n` to another value of the same type, with the same `n`. And

`Vec.zip` could require its two arguments to have the same length. Fixed-length vectors and matrices are also useful in mathematics.

Unfortunately, this more precise information comes at a cost. Dependent inductive types cause difficulties when the terms they depend on are provably equal but not syntactically equal up to computation (e.g., `m + n` vs. `n + m`). In this guide, we will generally avoid dependent inductive types. They are briefly covered in this section for completeness. To put it unambiguously: This is not exam material.

The definitions below introduce conversions between lists and vectors:

```
def listOfVec {α : Type} : ∀{n : ℕ}, Vec α n → List α
  | _, Vec.nil      => []
  | _, Vec.cons a v => a :: listOfVec v

def vecOfList {α : Type} :
  ∀xs : List α, Vec α (List.length xs)
  | []      => Vec.nil
  | x :: xs => Vec.cons x (vecOfList xs)
```

The `listOfVec` conversion takes a type `α`, a length `n`, and a vector of length `n` over `α` as arguments and returns a list over `α`. Although we do not care about the length `n`, it still needs to be an argument because it appears in the type of the third argument. We make the first two arguments, `α` and `n`, implicit since they can be inferred from the type of the third argument.

The `vecOfList` conversion takes a type `α` and a list over `α` as arguments and returns a vector of the same length as the list. Lean's type checker is strong enough to determine that the two right-hand sides have the desired type.

By the PAT principle, proofs are analogous to function definitions. Let us verify that converting a vector to a list preserves its length:

```
theorem length_listOfVec {α : Type} :
  ∀(n : ℕ) (v : Vec α n), List.length (listOfVec v) = n
  | _, Vec.nil      => by rfl
  | _, Vec.cons a v =>
    by simp [listOfVec, length_listOfVec _ v]
```

To prove a goal `v : Vec α n ⊢ P[v]` by structural induction on `v`, we might naively think that it suffices to show the following two subgoals:

```
                                  ⊢ P[Vec.nil]
m : ℕ, a : α, u : Vec α m, ih : P[u] ⊢ P[Vec.cons a u]
```

This is naive because the subgoals are not even type-correct: The hole in `P[ ]` has type `Vec α n` (the type of its original dweller, `v`), so we cannot simply plug `Vec.nil`, `u`, or `Vec.cons a u`—which have types `Vec α 0`, `Vec α m`, and `Vec α (m + 1)`—into that hole. We must massage `P` each time, replacing `n` with `0`, `m`, or `m + 1`. Using the notation $\texttt{P}_\texttt{t}\texttt{[ ]}$ for the variant of `P[ ]` where all occurrences of `n` are replaced by term `t`, we get

$$\begin{array}{r} \vdash \texttt{P}_0\texttt{[Vec.nil]} \\ \texttt{m} : \mathbb{N}, \texttt{a} : \alpha, \texttt{u} : \texttt{Vec}\ \alpha\ \texttt{m}, \texttt{ih} : \texttt{P}_\texttt{m}\texttt{[u]} \vdash \texttt{P}_{\texttt{m+1}}\texttt{[Vec.cons a u]} \end{array}$$

Proofs by case distinction using the `cases` tactic are similar, but without the induction hypothesis. Often, the length `n` will be not a variable but a complex

term. Then the replacement of `n` in `P[ ]` might not be intuitively meaningful. With `cases`, the corresponding subgoal is silently eliminated. Thus, a case distinction on a value of type `Vec α 0` will yield only one subgoal, of the form `⊢ P[Vec.nil]`, since `0` could never be equal to a term of the form `m + 1`.

Dependently typed pattern matching is subtle, because the type of the value we match on may change according to the constructor. Given `v : Vec α n`, we might be tempted to write

```
match v with
| Vec.nil      => ...
| Vec.cons a u => ...
```

but this is just as naive as our first induction proof attempt above. Since the term `n` in the type `Vec α n` may change depending on the constructor, we must pattern-match on `n` as well:

```
match n, v with
| 0,     Vec.nil      => ...
| m + 1, Vec.cons a u => ...
```

Showing the implicit arguments, we have

```
match n, v with
| 0,     @Vec.nil α        => ...
| m + 1, @Vec.cons α a m u => ...
```

Often, it is sufficient to put placeholders in the first column:

```
match n, v with
| _, Vec.nil      => ...
| _, Vec.cons a u => ...
```

It may seem paradoxical to pattern-match on `n` only to ignore the result, but without it Lean cannot infer the second implicit argument of `Vec.cons`. In this respect, `cases` is more user-friendly than `match`.

5.11 Summary of New Lean Constructs

Declarations

`class`	declares a structure type as a type class
`instance`	declares a structure value as a type class instance
`structure`	introduces a structure type and its selectors

Expressions

`if ... then ... else`	performs a case distinction on a decidable proposition
`match ... with`	performs pattern matching

Tactic

`cases`	performs a case distinction

Chapter 6

Inductive Predicates

Inductive predicates, or inductively defined propositions, are a convenient way to specify functions of type $\cdots \to \texttt{Prop}$. They are reminiscent of formal systems (Section 1.3) and of Prolog-style logic programming. But Lean offers a much stronger logic than Prolog, so we need to do some work to establish theorems instead of just running the Prolog interpreter. A possible view of Lean:

Lean = functional programming + logic programming + more logic

6.1 Introductory Examples

Unless you have been exposed to Prolog or logic programming, you will probably wonder what inductive predicates are and why they are useful. We start by reviewing three examples that demonstrate the variety of uses: even numbers, tennis games, and the reflexive transitive closure.

6.1.1 Even Numbers

Mathematicians often define sets as the smallest set that meets some criteria. Consider this definition:

> The set E of *even natural numbers* is defined as the smallest set S closed under the following rules:
>
> (1) $0 \in S$;
>
> (2) for every $k \in \mathbb{N}$, if $k \in S$, then $k + 2 \in S$.
>
> Such a set exists by the Knaster–Tarski theorem.

(The last sentence is often left implicit.) It is easy to convince ourselves that E contains all the even numbers and only those. Let us put on our mathematician's hat and prove that 4 is even:

> By rule (1), we have $0 \in E$.
>
> Hence, by rule (2) (with $k := 0$), we have $2 \in E$.
>
> Thus, by rule (2) (with $k := 2$), we have $4 \in E$, as desired. □

By contrast, computer scientists might use a formal system consisting of two derivation rules to specify the same set:

$$\frac{}{0 \in E}\,\textsc{Zero} \qquad\qquad \frac{k \in E}{k + 2 \in E}\,\textsc{AddTwo}_k$$

A proof is then a derivation tree:

$$\dfrac{\dfrac{\dfrac{}{0 \in E}\,\textsc{Zero}}{2 \in E}\,\textsc{AddTwo}_0}{4 \in E}\,\textsc{AddTwo}_2$$

The proof is forward if we read it downwards and backward if we read it upwards.

Inductive predicates are the logician's way to achieve the same result. In Lean, instead of a set, we would define a characteristic predicate inductively:

```
inductive Even : ℕ → Prop where
  | zero    : Even 0
  | add_two : ∀k : ℕ, Even k → Even (k + 2)
```

This should look familiar. We have used the same syntax, except with `Type` instead of `Prop`, to define inductive types. Inductive types and inductive predicates are provided by the same mechanism in Lean, in accordance with the PAT principle.

The command above defines a unary predicate `Even` as well as two introduction rules `Even.zero` and `Even.add_two` that can be used to prove goals of the form `⊢ Even ...`. Recall that an introduction rule for a symbol (e.g., `Even`) is a theorem whose conclusion contains that symbol (Section 4.3). By the PAT principle, `Even n` can be viewed as a dependent inductive type like `Vec α n` (Section 5.10), and `Even.zero` and `Even.add_two` as constructors like `Vec.nil` and `Vec.cons`.

If we translate the Lean definition back into English, we get something similar to the Knaster–Tarski-style definition above:

> The following clauses define the even numbers.
>
> (1) `0` is even;
>
> (2) any number of the form `k + 2` is even, assuming `k` is even.
>
> Any other number is not even.

It is worth nothing that inductively defined symbols have no definition. Thus, we cannot unfold their definition using `simp`, any more than we can unfold the definition of an inductive type such as `List α`. The only reasoning principles available are introduction and elimination.

As a warm-up exercise, here is a proof of `Even 4`:

```
theorem Even_4 :
  Even 4 :=
  have Even_0 : Even 0 :=
    Even.zero
  have Even_2 : Even 2 :=
```

```
    Even.add_two _ Even_0
  show Even 4 from
    Even.add_two _ Even_2
```

For example, the proof term `Even.add_two _ Even_0` has type `Even (0 + 2)`, which is syntactically equal to `Even 2` up to computation and hence equal from the type checker's point of view. The underscore stands for `0`.

Thanks to the "no junk" guarantee of inductive definitions, `Even.zero` and `Even.add_two` are the only two ways to construct proofs of `⊢ Even ...`. By inspecting the conclusions `Even 0` and `Even (k + 2)`, we see that there is no danger of ever proving that `1` is even.

Another way to view the inductive definition above is as follows: The first line introduces a predicate, whereas the second and third lines introduce axioms we want the predicate to satisfy. Accordingly, we could have written

```
opaque Even : ℕ → Prop
axiom Even.zero    : Even 0
axiom Even.add_two : ∀k : ℕ, Even k → Even (k + 2)
```

replacing `inductive` by `opaque` and `|` by `axiom`. But this axiomatic version, apart from being dangerous, does not give us any information about when `Even` is false. We cannot use it to prove `¬ Even 1` or `¬ Even 17`. For all we know, `Even` could be true for all natural numbers. In contrast, the inductive definition guarantees that we obtain the least (i.e., the most false) predicate that satisfies the introduction rules `Even.zero` and `Even.add_two`, and provides elimination and induction principles that allow us to prove `¬ Even 1`, `¬ Even 17`, or `¬ Even (2 * n + 1)`.

Why should we bother with inductive predicates when we can define recursive functions? Indeed, the following definition is perfectly legitimate:

```
def evenRec : ℕ → Bool
  | 0     => true
  | 1     => false
  | k + 2 => evenRec k
```

Each style has its strengths and weaknesses. The recursive version forces us to specify a false case (the second equation), and it forces us to worry about termination. On the other hand, because it is equational and computational, it works well with `rfl`, `simp`, `#eval`, and `#reduce`. The inductive version is arguably more abstract and elegant. Each introduction rule is stated independently. We can add or remove rules without having to think about termination or executability.

Yet another way to define `Even` is as a nonrecursive definition, using the modulo operator (`%`):

```
def evenNonrec (k : ℕ) : Prop :=
  k % 2 = 0
```

Mathematicians would likely prefer this version. But the inductive version is a convenient "Hello, World!" example that resembles many realistic inductive definitions. It might be a toy, but it is a useful toy.

6.1.2 Tennis Games

Transition systems consists of transition rules that connect a "before" and an "after" state. As a specimen of a transition system, we consider the possible transi-

tions in a game of tennis, starting from 0–0 ("Love all"). Tennis games are also a toy, but in Chapter 9, we will define the semantics of an imperative programming language as a transition system in a similar style.

The scoring rules for tennis from the International Tennis Federation's *Rules of Tennis* are reproduced below.

> A standard game is scored as follows with the server's score being called first:
>
> No point – "Love"
> First point – "15"
> Second point – "30"
> Third point – "40"
> Fourth point – "Game"
>
> except that if each player/team has won three points, the score is "Deuce." After "Deuce," the score is "Advantage" for the player/team who wins the next point. If that same player/team also wins the next point, that player/team wins the "Game"; if the opposing player/team wins the next point, the score is again "Deuce." A player/team needs to win two consecutive points immediately after "Deuce" to win the "Game."

We first define an inductive type to represent scores:

```
inductive Score : Type where
  | vs       : ℕ → ℕ → Score
  | advServ  : Score
  | advRecv  : Score
  | gameServ : Score
  | gameRecv : Score
```

A score such as 30–15 is represented as `Score.vs 30 15`, which we will also write in infix notation as `30–15`. We ignore some of the most frivolous aspects of the scoring rules, writing `0` for "Love" and `40–40` for "Deuce." If we really cared, we could introduce aliases such as `def love : ℕ := Score.vs 0 0`.

The next stage is to introduce a binary predicate `Step` that determines whether a transition is possible:

```
inductive Step : Score → Score → Prop where
  | serv_0_15     : ∀n, Step (0–n) (15–n)
  | serv_15_30    : ∀n, Step (15–n) (30–n)
  | serv_30_40    : ∀n, Step (30–n) (40–n)
  | serv_40_game  : ∀n, n < 40 → Step (40–n) Score.gameServ
  | serv_40_adv   : Step (40–40) Score.advServ
  | serv_adv_40   : Step Score.advServ (40–40)
  | serv_adv_game : Step Score.advServ Score.gameServ
  | recv_0_15     : ∀n, Step (n–0) (n–15)
  | recv_15_30    : ∀n, Step (n–15) (n–30)
  | recv_30_40    : ∀n, Step (n–30) (n–40)
  | recv_40_game  : ∀n, n < 40 → Step (n–40) Score.gameRecv
  | recv_40_adv   : Step (40–40) Score.advRecv
```

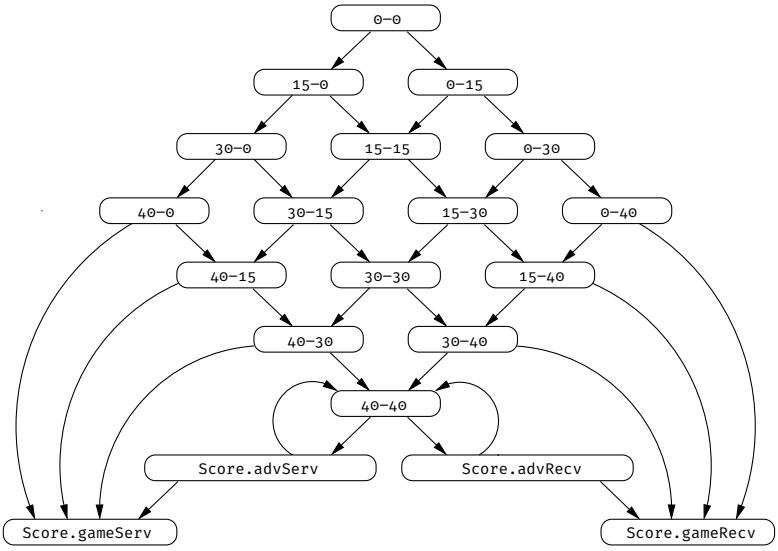
```
  | recv_adv_40   : Step Score.advRecv (40–40)
  | recv_adv_game : Step Score.advRecv Score.gameRecv
```

Let `s ⇝ t` abbreviate `Step s t`. A game is a chain $s_0 \rightsquigarrow s_1 \rightsquigarrow s_2 \rightsquigarrow \cdots \rightsquigarrow s_n$ where s_0 = 0–0 and no transition is possible from s_n. The predicate allows nonsensical transitions such as 15–99 ⇝ 30–99, but since the score 15–99 cannot be reached from 0–0, these transitions are harmless.

Equipped with a formal definition, we can ask, and formally answer, questions such as: Do games have a maximal length? How many different final scores are possible? Is the score 15–99 reachable from "Love all"? And can the score ever return to 0–0? Let us use Lean to disprove the last claim:

```
theorem no_Step_to_0_0 (s : Score) :
  ¬ s ⇝ 0–0 :=
  by
    intro h
    cases h
```

The diagram below summarizes which scores are reachable from which scores:



6.1.3 Reflexive Transitive Closure

As a non-toy application of inductive predicates, consider the reflexive transitive closure r^* of a binary relation r. Informally, r^* is the relation that represents zero or more steps of r. For example, if r corresponds to taking one transition in a transition system (e.g., an automaton), then r^* corresponds to taking any number of steps, including zero. For a more concrete example, consider the following:

$$r = \{(1,2),(2,4),(4,8)\}$$
$$r^* = \{(n,n) \mid n \in \mathbb{N}\} \cup \{(1,2),(1,4),(1,8),(2,4),(2,8),(4,8)\}$$

The star (*) operator is often defined rigorously as a formal system:

$$\frac{(a,b) \in r}{(a,b) \in r^*}\ \textsc{Base} \qquad \frac{}{(a,a) \in r^*}\ \textsc{Refl} \qquad \frac{(a,b) \in r^* \qquad (b,c) \in r^*}{(a,c) \in r^*}\ \textsc{Trans}$$

These rules define r^* as the smallest relation that contains r (by BASE) and that is reflexive (by REFL) and transitive (by TRANS). If we wanted to define the transitive closure r^+ instead, we would simply omit the REFL rule. If we wanted the reflexive symmetric closure, we would replace the TRANS rule by a SYMM rule. With a formal system, we simply declare the properties we want to be true, without giving a thought to termination or executability.

It is straightforward to translate the above derivation rules into introduction rules of an inductive predicate:

```
inductive Star {α : Type} (R : α → α → Prop) : α → α → Prop
where
  | base (a b : α)    : R a b → Star R a b
  | refl (a : α)      : Star R a a
  | trans (a b c : α) : Star R a b → Star R b c → Star R a c
```

We represent relations as binary predicates rather than sets of pairs, writing `R a b` for `(a, b) ∈ R`. Relations and predicates are interchangeable, but predicates are often more convenient to use in a proof assistant. The reflexive transitive closure of `R` is written `Star R`. Notice that `a`, `b`, and `c` are declared as parameters of the introduction rules, on the left of the colons. We could also have written

```
| base : ∀a b : α, R a b → Star R a b
```

or at the other extreme

```
| base (a b : α) (hab : R a b) : Star R a b
```

All these forms are logically and operationally equivalent.

The general format of inductive predicates is as follows:

```
inductive predicate-name (params₁ : type₁) ... (paramsₖ : typeₖ) :
  typeₖ₊₁ → ··· → typeₖ₊ₚ → Prop where
  | rule-name₁ (params₁₁ : type₁₁) ... (params₁ₘ₁ : type₁ₘ₁) :
    proposition₁
      ⋮
  | rule-nameₙ (paramsₙ₁ : typeₙ₁) ... (paramsₙₘₙ : typeₙₘₙ) :
    propositionₙ
```

where the conclusion of each $proposition_j$ must be an application of the defined predicate *predicate-name* to some arguments. These arguments may be arbitrary terms; they do not need to be constructor patterns. We can also use curly braces `{ }` instead of parentheses `( )` if we want to make the arguments corresponding to the parameters implicit.

The above definition of `Star` is truly elegant. If you still doubt this, try implementing it as a recursive function:

```
def starRec {α : Type} (R : α → α → Bool) :
  α → α → Bool :=
```

To summarize, each introduction rule of an inductive predicate `P` consists of the following components, from left to right:

- a *name*;
- *variables* that may appear in the rule;

- zero or more *conditions* that must be fulfilled, which may apply `P` recursively;
- an application of `P` to some arguments, forming a *pattern*.

Accordingly, for rule `Star.base`, the pattern is `Star R a b`, the condition is `R a b`, and the variables are `a` and `b`.

6.1.4 A Nonexample

Not all inductive definitions admit a least solution. The simplest nonexample is

```
-- fails
inductive Illegal : Prop where
  | intro : ¬ Illegal → Illegal
```

If Lean accepted this definition, we could use it to prove the equivalence `Illegal ↔ ¬ Illegal`, from which we could easily derive `False`. Fortunately, Lean rejects the definition:

```
arg #1 of 'Illegal.intro' has a non positive occurrence of
the datatypes being declared
```

The nonpositive occurrence it complains about is the occurrence of `Illegal` under a negation. Mathematicians would reject the definition on the ground that the monotonicity condition of the Knaster–Tarski theorem is not satisfied.

6.2 Logical Symbols

Although `Even` is the first openly inductive predicate in this guide, the earlier chapters already presented other inductive predicates clandestinely. The very first of these is equality (`=`), introduced in Chapter 2, followed by the logical symbols `∧`, `∨`, `↔`, `∃`, `True`, and `False`. Their definitions are worth studying closely:

```
inductive And (a b : Prop) : Prop where
  | intro : a → b → And a b

inductive Or (a b : Prop) : Prop where
  | inl : a → Or a b
  | inr : b → Or a b

inductive Iff (a b : Prop) : Prop where
  | intro : (a → b) → (b → a) → Iff a b

inductive Exists {α : Type} (P : α → Prop) : Prop where
  | intro : ∀a : α, P a → Exists P

inductive True : Prop where
  | intro : True

inductive False : Prop where

inductive Eq {α : Type} : α → α → Prop where
  | refl : ∀a : α, Eq a a
```

(Strictly speaking, in Lean, some of the above definitions are actually structures and not inductive predicates, but the difference is superficial. As we saw in Section 5.5, structures are essentially single-constructor inductive predicates with some syntactic sugar.)

The traditional notations `∃x : α, P` and `x = y` are syntactic sugar for `Exists (fun x : α ↦ P)` and `Eq x y`. Notice how `fun` acts as an all-purpose binder. Notice also that there is no constructor for `False`. There are no proofs of `False`, just like there is no proof of `Even 1`. With inductive predicates, we state only the rules we want to be true.

The symbol `∀`, including its special case `→`, is built directly into the logic and is not defined as an inductive predicate. Nor does it have explicit introduction or elimination rules. The introduction principle is the anonymous function `fun x ↦ _`, and the elimination principle is function application `_ u`.

As for any inductive predicates, only the introduction rules are specified. The elimination rules presented in Sections 3.3 and 3.4 must be derived manually.

6.3 Rule Induction

In the same way that we can perform an induction on a term of inductive type, we can perform an induction on a proof of an inductive predicate. For example, given the goal `h : Even n ⊢ P n`, we can invoke `induction h` and get two subgoals, for `Even.zero` and `Even.add_two`. This is called *induction on the structure of the derivation of* `h` or simply *rule induction*, because the induction is on the predicate's introduction rules (i.e., the constructors of the proof term).

There are two ways to look at rule induction: the "least-predicate-such-that view" and the "PAT view." To understand the least-predicate-such-that view, recall that an inductive definition introduces a symbol as the least (i.e., the most false) predicate satisfying the introduction rules. Accordingly, `Even` is the least predicate `Q` such that the properties `Q 0` and `∀k, Q k → Q (k + 2)` hold. Therefore, if we can show that `P 0` and `∀k, P k → P (k + 2)` hold for some predicate `P`, then `P` is either `Even` itself or greater than (i.e., more true than) `Even`. As a result, `Even n` implies `P n`, which is exactly what we need to prove the goal `h : Even n ⊢ P n`.

The least-predicate-such-that view gives a nice intuitive account of rule induction that can be used in informal arguments, such as the following proof that `Even n` implies `n % 2 = 0` for all `n`:

> The proof is by rule induction on the hypothesis `Even n`.
>
> CASE `Even.zero`: We must show `0 % 2 = 0`. This follows by computation.
>
> CASE `Even.add_two k`: The induction hypothesis is `k % 2 = 0`. We must show `(k + 2) % 2 = 0`. This follows by basic arithmetic reasoning. □

The Lean proof has the same structure:

```
theorem mod_two_Eq_zero_of_Even (n : ℕ) (h : Even n) :
  n % 2 = 0 :=
  by
    induction h with
    | zero            => rfl
    | add_two k hk ih => simp [ih]
```

The PAT principle gives us another fruitful way to look at rule induction. The key idea is that rule induction on `h` in a goal such as `h : Even n ⊢ P[h]` is perfectly analogous to structural induction on a value of a dependent inductive type such as `Vec α n` (Section 5.10). Writing $\texttt{P}_\texttt{u}\texttt{[ ]}$ for the variant of `P[ ]` where `n` is replaced by some term `u`, we get the subgoals

$$\vdash \texttt{P}_0\texttt{[Even.zero : Even 0]}$$
$$\texttt{k} : \mathbb{N}, \texttt{hk} : \texttt{Even k}, \texttt{ih} : \texttt{P}_\texttt{k}\texttt{[hk]} \vdash \texttt{P}_{\texttt{k+2}}\texttt{[Even.add\_two k hk : Even (k + 2)]}$$

These are effectively the subgoals produced by the `induction` tactic.

Regardless of the inductive predicate `Q`, the procedure to compute the subgoals is always the same:

1. Replace `h` in `P[h]` with each possible introduction rule applied to fresh variables (e.g., `Even.add_two k hk`), instantiating `n` in `P[ ]` to make `P[ ]` type-correct. This yields as many subgoals as there are introduction rules.
2. Add these new variables (e.g., `k`, `hk`) to the local context.
3. Add induction hypotheses for all new hypotheses that assert `Q` ....

Notice the presence of `hk : Even k` among the hypotheses above. It was absent in the least-predicate-such-that view and is not essential because it can always be recovered by strengthening `P` to be of the form `Even n ∧ ···`.

In nearly all practical cases, `h` will not occur in `P[h]`. We can then simply write

$$\vdash \texttt{P}_0 \qquad\qquad \texttt{k} : \mathbb{N}, \texttt{hk} : \texttt{Even k}, \texttt{ih} : \texttt{P}_\texttt{k} \vdash \texttt{P}_{\texttt{k+2}}$$

In rare cases, `h` will occur in `P[h]`. Proofs may appear as subterms in arbitrary terms, as we saw when we tried to extract the head of a list in Section 5.7.

The reflexive transitive closure `Star R` is similar to `Even`. Given a goal `h : Star R x y ⊢ P`, rule induction on `h` produces the following subgoals, where $\texttt{P}_{\texttt{t,u}}$ denotes the variant of `P` where `x` and `y` are replaced by `t` and `u`, respectively:

$$\texttt{a b} : \alpha, \texttt{hab} : \texttt{R a b} \vdash \texttt{P}_{\texttt{a,b}}$$
$$\texttt{a} : \alpha \vdash \texttt{P}_{\texttt{a,a}}$$
$$\texttt{a b c} : \alpha, \texttt{hab} : \texttt{Star R a b}, \texttt{hbc} : \texttt{Star R b c}, \texttt{ihab} : \texttt{P}_{\texttt{a,b}}, \texttt{ihbc} : \texttt{P}_{\texttt{b,c}} \vdash \texttt{P}_{\texttt{a,c}}$$

This is where the "assistant" aspect of "proof assistant" comes into play. One of the key properties of `Star` is idempotence—applying `Star` to `Star R` has no effect. This can be proved as follows in Lean, using rule induction for the $\to$ direction of the equivalence:

```
theorem Star_Star_Iff_Star {α : Type} (R : α → α → Prop)
    (a b : α) :
  Star (Star R) a b ↔ Star R a b :=
  by
    apply Iff.intro
    { intro h
      induction h with
      | base a b hab                    => exact hab
      | refl a                          => apply Star.refl
      | trans a b c hab hbc ihab ihbc =>
```

```
      apply Star.trans a b
      { exact ihab }
      { exact ihbc } }
  { intro h
    apply Star.base
    exact h }
```

We are careful to give intuitive names to the emerging variables. It is easy to get lost in goals containing long, automatically generated names. The cleanup tactics introduced in Section 3.8 can also be of help when we face large goals.

We can state the idempotence property more standardly in terms of equality instead of as an equivalence:

```
@[simp] theorem Star_Star_Eq_Star {α : Type}
    (R : α → α → Prop) :
  Star (Star R) = Star R :=
  by
    apply funext
    intro a
    apply funext
    intro b
    apply propext
    apply Star_Star_Iff_Star
```

The proof uses two theorems that are available because Lean's logic is classical:

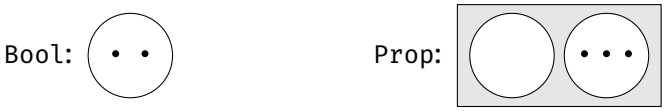
```
funext : (∀x, ?f x = ?g x) → ?f = ?g
propext : (?a ↔ ?b) → ?a = ?b
```

Functional extensionality (`funext`) states that if two functions yield equal results for all inputs, then the two functions must be equal. *Propositional extensionality* (`propext`) states that equivalence of propositions coincides with equality. In these phrases, extensionality means something like "what you see is what you get." These properties may seem obvious, and yet there exist proof assistants built on weaker, intuitionistic logics in which the properties do not generally hold.

We register the theorem `Star_Star_Eq_Star` as a simp rule, because viewed as a left-to-right rewrite rule, it genuinely replaces a complex term by a simpler term. It is hard to imagine a situation where we would not want `simp` to rewrite `Star (Star ...)` to `Star ...`.

For rule induction, we use the `induction` tactic. For subtle logical reasons that will become clearer in Chapter 12, rule induction by pattern matching and recursion is not allowed.

In Section 5.4, we saw a diagram depicting the interpretation of `Bool` and `Prop` side by side. The diagram suggested the existence of an infinite number of propositions, but we now know that there are exactly two propositions: `False` and `True`. Here is a revised diagram:



We see that there are only two propositions, one which has no proofs and one which has a number of proofs. Three proofs are shown in the diagram.

6.4 Linear Arithmetic Tactic

linarith

The `linarith` tactic can be used to prove goals involving linear arithmetic equalities (`=`), inequalities (`<`, `>`, $\leq$, and $\geq$), and disequalities ($\neq$). *Linear* means that multiplication and division do not occur, or if they do then one of the operands must be a numeric constant. For example, `2 * x < y` is a linear constraint (which can be rewritten to `x + x < y`), whereas `x * y < y` is nonlinear.

6.5 Elimination

Given an inductive predicate `Q`, its introduction rules typically are of the form $\forall \ldots, \cdots \to$ `Q ...` and can be used to prove goals of the form $\vdash$ `Q ...`. Elimination works the other way around: It extracts information from a theorem or hypothesis `h : Q ...`. Elimination takes many forms: the `cases` and `induction` tactics, pattern matching, and elimination rules (e.g., `And.left`).

Invoked on `h : Q ...`, the `cases h` tactic performs roughly the same rule induction as `induction h` but without producing any induction hypotheses. We encountered two idioms in Chapter 5 that we can finally analyze.

The first idiom is when `h` is of the form `l = r`—i.e., `Eq l r` (Section 6.2). Suppose the goal is `h : l = r` $\vdash$ `P[h]`. The procedure presented in Section 6.3 produces the subgoal

$$\texttt{a} : \alpha \vdash \texttt{P}_{\texttt{a},\texttt{a}}\texttt{[Eq.refl a : a = a]}$$

in which $\texttt{P}_{\texttt{t},\texttt{u}}\texttt{[ ]}$ stands for the variant of `P[ ]` where `l` and `r` are replaced by `t` and `u`, respectively. (Strictly speaking, the useless hypothesis `h : a = a` would also appear in the subgoal.) In practice, `P[h]` would likely not depend on `h`. Moreover, `cases` reuses the name `l` instead of using a different name like `a`. Thus, we would get

$$\texttt{l} : \alpha \vdash \texttt{P}_{\texttt{l},\texttt{l}}$$

In other words, all occurrences of `r` in the original goal have been replaced by `l`. This corresponds to the behavior we observed in Section 5.7.

The second idiom is the tactic `cases Classical.em Q`, where `Q` is a proposition. The `Classical.em Q` part is a proof term for `Q` $\vee \neg$ `Q`—i.e., `Or Q (¬ Q)` (Section 6.2). Then `cases` is applied to eliminate the $\vee$ connective. Suppose the goal is $\vdash$ `P[Classical.em Q]`. By the definition of the `Or` predicate, the new subgoals are

$$\texttt{hQ} : \texttt{Q} \vdash \texttt{P[Or.inl hQ : Q} \vee \neg \texttt{Q]}$$
$$\texttt{hnQ} : \neg \texttt{Q} \vdash \texttt{P[Or.inr hnQ : Q} \vee \neg \texttt{Q]}$$

There is no need to modify `P` because `Or.inl hQ` and `Or.inr hnQ` have the same type as `Classical.em Q`—namely, `Q` $\vee \neg$ `Q`. In practice, the goal would usually not contain `Classical.em Q` and be simply $\vdash$ `P`, and we would then have the subgoals

$$\texttt{hQ} : \texttt{Q} \vdash \texttt{P} \qquad\qquad \texttt{hnQ} : \neg \texttt{Q} \vdash \texttt{P}$$

Again, this is the behavior we observed in Section 5.7.

In structured proofs, we can use `match` expressions (Section 5.4) to achieve the same effect as `cases`. This works well for logical symbols. However, for predicates

such as `Even` and `Star`, with arguments that evolve through the induction, we end up with dependently typed pattern matching, which is subtle (Section 5.10). It is generally easier to let `cases` figure out what the subgoals should look like than to pattern-match. We will review an example of both styles below.

It can be useful to expand a hypothesis of the form `Q (c ...)`, where `c` is a constructor or some other constant. We can state and prove an *inversion rule* to support such eliminative reasoning. A typical inversion rule has the form

$$\forall x_1 \ldots x_n,\ Q\ (c\ x_1 \ldots x_n) \to (\exists \ldots, \cdots \wedge \cdots) \vee \cdots \vee (\exists \ldots, \cdots \wedge \cdots)$$

It can be useful to combine introduction and elimination into a single theorem, which can be used for rewriting both the hypotheses and the targets of goals. The format is the same except for the connective $\leftrightarrow$ in the middle:

$$\forall x_1 \ldots x_n,\ Q\ (c\ x_1 \ldots x_n) \leftrightarrow (\exists \ldots, \cdots \wedge \cdots) \vee \cdots \vee (\exists \ldots, \cdots \wedge \cdots)$$

An inversion rule for `Even` is given below:

```
theorem Even_Iff (n : ℕ) :
  Even n ↔ n = 0 ∨ (∃m : ℕ, n = m + 2 ∧ Even m) :=
  by
    apply Iff.intro
    { intro hn
      cases hn with
      | zero         => simp
      | add_two k hk =>
        apply Or.inr
        apply Exists.intro k
        simp [hk] }
    { intro hor
      cases hor with
      | inl heq => simp [heq, Even.zero]
      | inr hex =>
        cases hex with
        | intro k hand =>
          cases hand with
          | intro heq hk =>
            simp [heq, Even.add_two _ hk] }
```

As usual, the tactical proof is not particularly readable, but we see that introduction rules and the eliminative `cases` tactic play a major role, for both the logical symbols and the `Even` predicate. The `simp` tactic puts the final touches.

If you prefer structured proofs, here is a version of the proof with dependently typed pattern matching on `hn : Even n`:

```
theorem Even_Iff_struct (n : ℕ) :
  Even n ↔ n = 0 ∨ (∃m : ℕ, n = m + 2 ∧ Even m) :=
  Iff.intro
    (assume hn : Even n
     match n, hn with
     | _, Even.zero         =>
       show 0 = 0 ∨ _ from
```

```
      by simp
  | _, Even.add_two k hk =>
    show _ ∨ (∃m, k + 2 = m + 2 ∧ Even m) from
      Or.inr (Exists.intro k (by simp [*])))
(assume hor : n = 0 ∨ (∃m, n = m + 2 ∧ Even m)
 match hor with
 | Or.inl heq =>
   show Even n from
     by simp [heq, Even.zero]
 | Or.inr hex =>
   match hex with
   | Exists.intro m hand =>
     match hand with
     | And.intro heq hm =>
       show Even n from
         by simp [heq, Even.add_two _ hm])
```

6.6 Further Examples

Equipped with a better understanding of inductive predicates, we are now ready to review four further applications in turn.

6.6.1 Sorted Lists

Our first example is a predicate that checks whether a list of natural numbers is sorted in increasing order:

```
inductive Sorted : List ℕ → Prop where
  | nil : Sorted []
  | single (x : ℕ) : Sorted [x]
  | two_or_more (x y : ℕ) {zs : List ℕ} (hle : x ≤ y)
      (hsorted : Sorted (y :: zs)) :
    Sorted (x :: y :: zs)
```

This definition captures the following mathematical intuition:

> The set of sorted lists is defined as the smallest set closed under the following rules:
>
> (1) the list `[]` is sorted;
>
> (2) given a number `x`, the list `[x]` is sorted;
>
> (3) given two numbers `x`, `y` and a list `zs`, if `x < y` and `y :: zs` is sorted, then `x :: y :: zs` is sorted.

It is always a good idea to test our definitions by trying it on small examples. Is the list `[3, 5]` sorted? It would appear so:

```
theorem Sorted_3_5 :
  Sorted [3, 5] :=
  by
```

```
    apply Sorted.two_or_more
    { simp }
    { exact Sorted.single _ }
```

The example needs two of the introduction rules for `sorted`. This approach tends to work well for expressions consisting only of closed terms. A more compact proof follows, using proof terms:

```
theorem Sorted_3_5_raw :
  Sorted [3, 5] :=
  Sorted.two_or_more _ _ (by simp) (Sorted.single _)
```

The same idea can be used to prove that `[7, 9, 9, 11]` is sorted:

```
theorem sorted_7_9_9_11 :
  Sorted [7, 9, 9, 11] :=
  Sorted.two_or_more _ _ (by simp)
    (Sorted.two_or_more _ _ (by simp)
       (Sorted.two_or_more _ _ (by simp)
          (Sorted.single _)))
```

Conversely, we can show that some lists are not sorted. For this, we need elimination:

```
theorem Not_Sorted_17_13 :
  ¬ Sorted [17, 13] :=
  by
    intro h
    cases h with
    | two_or_more _ _ hlet hsorted => simp at hlet
```

From the hypothesis that the list `[17, 13]` is sorted, we extract the inequality $17 \leq 13$. The `cases` tactic silently eliminates the `nil` and `single` cases, because they cannot match a two-element list. Then we invoke `simp` to exploit the impossible hypothesis $17 \leq 13$.

6.6.2 Palindromes

Palindromes are lists that read the same from left to right and from right to left. For example, `[a, b, b, a]` and `[a, h, a]` are palindromes. The following inductive predicate is `True` if and only if the list passed as argument is a palindrome:

```
inductive Palindrome {α : Type} : List α → Prop where
  | nil : Palindrome []
  | single (x : α) : Palindrome [x]
  | sandwich (x : α) (xs : List α) (hxs : Palindrome xs) :
    Palindrome ([x] ++ xs ++ [x])
```

The definition distinguishes three cases: (1) `[]` is a palindrome; (2) for any element `x`, the singleton list `[x]` is a palindrome; (3) for any element `x` and any palindrome $[y_1, \ldots, y_n]$, the list $[x, y_1, \ldots, y_n, x]$ is a palindrome.

Palindromes are another example where inductive predicates come into their own. The following naive recursive definition cannot work because `[x] ++ xs ++ [x]` is not a constructor pattern, and the variable `x` is repeated:

```
-- fails
def palindromeRec {α : Type} : List α → Bool
  | []                => true
  | [_]               => true
  | ([x] ++ xs ++ [x]) => palindromeRec xs
  | _                 => false
```

A correct recursive definition is possible but beyond the scope of this guide.

Naturally, the reverse of a palindrome is a palindrome. It is a good exercise:

```
theorem Palindrome_reverse {α : Type} (xs : List α)
    (hxs : Palindrome xs) :
  Palindrome (reverse xs) :=
  by
    induction hxs with
    | nil                 => exact Palindrome.nil
    | single x            => exact Palindrome.single x
    | sandwich x xs hxs ih =>
      { simp [reverse, reverse_append]
        exact Palindrome.sandwich _ _ ih }
```

Informally:

> The proof is by rule induction on the hypothesis `hxs`.
>
> Case `Palindrome.nil`: We must show `Palindrome (reverse [])`. This follows from `Palindrome.nil` using `reverse [] = []`.
>
> Case `Palindrome.single x`: We must show `Palindrome (reverse [x])`. This follows from `Palindrome.single` using `reverse [x] = [x]`.
>
> Case `Palindrome.sandwich x xs hxs`: We must show `Palindrome (reverse ([x] ++ xs ++ [x]))` under the hypothesis (`hxs`) `Palindrome xs`. The induction hypothesis is `Palindrome (reverse xs)`. By simplification, it suffices to show `Palindrome ([x] ++ reverse xs ++ [x])`. By `Palindrome.sandwich`, it suffices to show `Palindrome (reverse xs)`, which is exactly the induction hypothesis. □

6.6.3 Full Binary Trees

Our third example is based on the type of binary trees introduced in Section 5.8:

```
inductive BTree (α : Type) : Type where
  | empty : BTree α
  | node  : α → BTree α → BTree α → BTree α
```

A binary tree is *full* if all its nodes have either zero or two children. This can be encoded as an inductive predicate:

```
inductive IsFull {α : Type} : BTree α → Prop where
  | empty : IsFull BTree.empty
  | node (a : α) (l r : BTree α)
      (hl : IsFull l) (hr : IsFull r)
      (hiff : l = BTree.empty ↔ r = BTree.empty) :
    IsFull (BTree.node a l r)
```

The first case states that the empty tree is a full tree. The second case states that a nonempty tree is a full tree if it has two child trees that are themselves full and that are both empty or both nonempty. The two cases neatly follow the structure of the inductive type, so it is natural to reuse the names `empty` and `node`.

The tree that consists of a node with empty trees as children is a full tree. Here is a simple proof:

```
theorem IsFull_singleton {α : Type} (a : α) :
  IsFull (BTree.node a BTree.empty BTree.empty) :=
  by
    apply IsFull.node
    { exact IsFull.empty }
    { exact IsFull.empty }
    { rfl }
```

A somewhat more interesting property of full trees is that fullness is preserved by the mirror operation. Our first proof is by rule induction on `ht : IsFull t`:

```
theorem IsFull_mirror {α : Type} (t : BTree α)
    (ht : IsFull t) :
  IsFull (mirror t) :=
  by
    induction ht with
    | empty                              => exact IsFull.empty
    | node a l r hl hr hiff ih_l ih_r =>
      { rw [mirror]
        apply IsFull.node
        { exact ih_r }
        { exact ih_l }
        { simp [mirror_Eq_empty_Iff, *] } }
```

Since `IsFull`'s definition follows `BTree`'s definition, it is also reasonable to perform structural induction on the tree `t`:

```
theorem IsFull_mirror_struct_induct {α : Type} (t : BTree α) :
  IsFull t → IsFull (mirror t) :=
  by
    induction t with
    | empty                 =>
      { intro ht
        exact ht }
    | node a l r ih_l ih_r =>
      { intro ht
        cases ht with
        | node _ _ _ hl hr hiff =>
          { rw [mirror]
            apply IsFull.node
            { exact ih_r hr }
            { apply ih_l hl }
            { simp [mirror_Eq_empty_Iff, *] } } }
```

The key is the case distinction on the hypothesis `ht : IsFull (BTree.node a l r)`. The `cases` tactic notices that the `IsFull.empty` introduction rule cannot have

been used to derive `ht`, so it only produces one case, corresponding to `IsFull.node`. As usual, the tactical proof will make more sense if you inspect it in Visual Studio Code, moving the cursor around.

6.6.4 First-Order Terms

Our last example is based on an inductive type of first-order terms:

```
inductive Term (α β : Type) : Type where
  | var : β → Term α β
  | fn  : α → List (Term α β) → Term α β
```

A first-order term is either a variable x or a function symbol f applied to a list of arguments: $f(t_1, \ldots, t_n)$, where the mathematical variables $t_1, \ldots, t_n$ stand for the arguments, which are themselves terms. Thus, $\sin(\max(x, y))$ is a first-order term. The parameters α and β are the types of function symbols and variables, respectively.

Not all terms are legal. For example, the term $\min(\cos(a), \cos(a, b))$ is considered ill-formed, because the function cos is invoked with inconsistent number of arguments (1 versus 2). Along with α and β, we also consider the arity, represented by a function `arity` : $\alpha \to \mathbb{N}$ indicating how many arguments each function symbol takes. For example, a binary symbol has arity 2.

The `WellFormed` predicate then checks whether the given term only contains function symbol applications with the specified number of arguments:

```
inductive WellFormed {α β : Type} (arity : α → ℕ) :
  Term α β → Prop where
  | var (x : β) : WellFormed arity (Term.var x)
  | fn (f : α) (ts : List (Term α β))
      (hargs : ∀t ∈ ts, WellFormed arity t)
      (hlen : length ts = arity f) :
    WellFormed arity (Term.fn f ts)
```

The `fn` case checks that the arguments `ts` are recursively well formed and that the length of `ts` equals the specified arity for the function symbol `f` in question.

Another interesting property of first-order terms is whether they contain variables. This can be checked easily using an inductive predicate:

```
inductive VariableFree {α β : Type} : Term α β → Prop where
  | fn (f : α) (ts : List (Term α β))
      (hargs : ∀t ∈ ts, VariableFree t) :
    VariableFree (Term.fn f ts)
```

There is no introduction rule corresponding to `Term.var` because variables are never variable-free.

6.7 Induction Pitfalls

Some care is needed when invoking `induction` on inductive predicates. The arguments of inductive predicates often evolve through the induction. Such details are often glossed over in informal proofs, but proof assistants require us to be precise.

Recall the definition of even numbers:

```
inductive Even : ℕ → Prop where
  | zero    : Even 0
  | add_two : ∀k : ℕ, Even k → Even (k + 2)
```

If the goal has the form `h : Even n ⊢ P n`, applying `induction` on `h` will produce the following subgoals:

```
⊢ P 0                    k : ℕ, hk : P k ⊢ P (k + 2)
```

This works as desired.

The problem is when `Even`'s argument is not a variable. Applying `induction` on the hypothesis `hev` in the goal `hev : Even (2 * n + 1) ⊢ False` fails with an error:

```
index in target's type is not a variable (consider using
the 'cases' tactic instead)
```

To solve this issue, we need to replace `2 * n + 1` by a variable `m` and add an equation `m = 2 * n + 1` as a hypothesis:

```
m : 2 * n + 1, hev : Even m ⊢ False
```

This goal is logically equivalent, but now `induction` produces two subgoals:

```
                                  m n : ℕ, hm : 0 = 2 * n + 1 ⊢ False
m : 2 * n + 1, ih : m = 2 * n + 1 → False, hm : m + 2 = 2 * n + 1 ⊢ False
```

Sadly, the second subgoal is not provable. The problem is that we want to instantiate the `n` in the induction hypothesis `ih` with `n - 1` but `n` is not instantiable.

The solution? We need to quantify over `n` to be able to instantiate it in the induction hypothesis. This is done by specifying `generalizing n` after `induction h`. Now we obtain the subgoals

```
                                  m n : ℕ, hm : 0 = 2 * n + 1 ⊢ False
m : 2 * n + 1, ih : ∀n, m = 2 * n + 1 → False, hm : m + 2 = 2 * n + 1 ⊢ False
```

Now the variable `n` in the conclusion is disconnected from the variable `n` in the induction hypothesis, and we can instantiate the hypothesis's `n` with the conclusion's `n - 1`.

Putting all of this together, we obtain

```
theorem Not_Even_two_mul_add_one (m n : ℕ)
    (hm : m = 2 * n + 1) :
  ¬ Even m :=
  by
    intro h
    induction h generalizing n with
    | zero            => linarith
    | add_two k hk ih =>
      apply ih (n - 1)
      cases n with
      | zero    => simp [Nat.ctor_eq_zero] at *
      | succ n' =>
        simp [Nat.succ_eq_add_one] at *
        linarith
```

We use the theorem `Nat.succ_eq_add_one` to rewrite terms of the form `Nat.succ n` to `n + 1` and the `linarith` tactic to perform simple arithmetic reasoning. The theorem is very useful when we face a mixture of `Nat.succ` and addition.

6.8 Summary of New Lean Constructs

Theorems

`funext`	functional extensionality
`propext`	propositional extensionality

Tactic

`linarith`	invokes a procedure for linear arithmetic

Chapter 7

Effectful Programming

Pure functional programming can sometimes feel overly restrictive. *Effectful functional programming* provides idioms that alleviate some of these restrictions, giving us the impression of programming with side effects, exceptions, nondeterminism, and other effects.

The underlying abstraction is called *monad*. Monads generalize programs with effects. They are popular in Haskell to write imperative programs. In Lean, they are used to express imperative programs and to reason about them. They are even useful for programming Lean itself, as we will see in Chapter 8.

These notes are inspired by Chapter 7 of *Programming in Lean* [2]. We also refer to Chapter 14 of *Real World Haskell* [24] for a general introduction to effectful functional programming.

7.1 Introductory Example

Consider the following programming task:

> Implement a function `sum257 ns` that sums up the second, fifth, and seventh items of a list `ns` of natural numbers. Use `Option ℕ` for the result so that you can return `Option.none` if the list has too few items.

A straightforward solution would be as follows:

```
def nth {α : Type} : List α → Nat → Option α
  | [],      _     => Option.none
  | x :: _,  0     => Option.some x
  | _ :: xs, n + 1 => nth xs n

def sum257 (ns : List ℕ) : Option ℕ :=
  match nth ns 1 with
  | Option.none    => Option.none
  | Option.some n₂ =>
    match nth ns 4 with
    | Option.none    => Option.none
    | Option.some n₅ =>
      match nth ns 6 with
      | Option.none    => Option.none
      | Option.some n₇ => Option.some (n₂ + n₅ + n₇)
```

(Confusingly, `nth` counts elements from 0.) The code is quite inelegant, because of all the pattern matching on `Option`. Although the programming task is contrived, we can all recall writing code with nested error handling and ever increasing indentation levels.

We can do better, by concentrating the ugliness in one function:

```
def connect {α : Type} {β : Type} :
  Option α → (α → Option β) → Option β
  | Option.none,   _ => Option.none
  | Option.some a, f => f a
```

The `connect` function works on an `Option`. If the value is `Option.none`, we leave it as is. This corresponds to an error condition, and errors are "sticky." Otherwise, the value is of the form `Option.some a`, and we apply the operation `f` on `a`—or *bind* `f`'s argument to `a`. We can now use `connect` to program our sum function:

```
def sum257Connect (ns : List ℕ) : Option ℕ :=
  connect (nth ns 1)
    (fun n₂ ↦ connect (nth ns 4)
       (fun n₅ ↦ connect (nth ns 6)
          (fun n₇ ↦ Option.some (n₂ + n₅ + n₇))))
```

Intuitively, the program performs the following steps:

1. Extract the second item from the list using `nth`. If there is no such item, `nth` returns `Option.none`; simply return this value. Otherwise, bind `n2` to this item and continue with the next step.
2. Perform the same for the fifth and seventh item, mutatis mutandis.
3. Return the sum of `n2`, `n5`, and `n7` in an `Option.some` wrapper.

Mathematically, our new function `sum257Connect` is equal to the original function `sum257`.

Instead of defining `connect` ourselves, we could have used Lean's predefined general `bind` operation. It takes the same arguments in the same order. Here is the new code:

```
def sum257Bind (ns : List ℕ) : Option ℕ :=
  bind (nth ns 1)
    (fun n₂ ↦ bind (nth ns 4)
       (fun n₅ ↦ bind (nth ns 6)
          (fun n₇ ↦ pure (n₂ + n₅ + n₇))))
```

We also use the predefined `pure` function instead of `Option.some` to convert a pure α value to an `Option` α.

One of the advantages of using the predefined `bind` is that it provides syntactic sugar, in the form of the `>>=` operator:

```
def sum257Op (ns : List ℕ) : Option ℕ :=
  nth ns 1 >>=
    fun n₂ ↦ nth ns 4 >>=
      fun n₅ ↦ nth ns 6 >>=
        fun n₇ ↦ pure (n₂ + n₅ + n₇)
```

The syntax `oa >>= f` expands to `bind oa f`, where `oa` is of type `Option` α.

The next-to-last version of the sum program uses heavier syntactic sugar:

```
def sum257Dos (ns : List ℕ) : Option ℕ :=
  do
    let n₂ ← nth ns 1
    do
      let n₅ ← nth ns 4
      do
        let n₇ ← nth ns 6
        pure (n₂ + n₅ + n₇)
```

The `do` notation provides a convenient syntax for effectful programs. The program `do let a ← oa ...` is equivalent to `oa >>= (fun a ↦ ...)`. If we are not interested in the result of `oa`'s computation, we can omit the `let a ←` binding and write `do oa ...`, which expands to `oa >>= (fun _ ↦ ...)`.

The `do` notation conveniently allows multiple `let` bindings in a single block. This brings us to the final version of the program:

```
def sum257Do (ns : List ℕ) : Option ℕ :=
  do
    let n₂ ← nth ns 1
    let n₅ ← nth ns 4
    let n₇ ← nth ns 6
    pure (n₂ + n₅ + n₇)
```

Each line with an arrow ← attempts to read a value. In case of failure, the entire program evaluates to `Option.none`.

The above function can be read as an imperative program where each of the `nth` calls can throw an exception. But even though the notation has an imperative flavor, the function is a pure functional program.

7.2 Two Operations and Three Laws

The `Option` type constructor is an example of a monad, called the *option monad*. In general, a monad is a unary type constructor `m : Type → Type` that depends on some type parameter α equipped with two distinguished operations:

```
pure {α : Type} : α → m α
bind {α β : Type} : m α → (α → m β) → m β
```

As usual, curly braces denote implicit arguments. For options, the `pure` operation is simply `Option.some`, whereas `bind` amounts to our `connect`.

A value of type `m α` is an effectful program. The `pure` operation embeds a pure, effectless program of type α in `m α`. The `bind` operation composes two effectful programs, of types `m α` and `m β`. The first program's output, of type α, is passed to the second program. The second program's output is also the output of the composite program.

We can think of a monad as a box containing some data. The box captures some special effect (e.g., exceptions, a mutable state). The `pure` operation puts data into the box, whereas `bind` allows us to access the data in the box and modify it—possibly even changing its type, since the result has type `m β`, not `m α`. There is, however, no general way to extract the data from the box—i.e., to obtain an α from an `m α`. There might not be any α value in it, or there might be several.

To summarize, `pure a` is a box containing the value `a`, with no effects, whereas `bind ma f` (also written `ma >>= f` or `do a ← ma, f a`) executes `ma`, then executes `f` with the unboxed result `a` of `ma`. It is convenient to use names such as `ma` or `mb` for boxed values of type `m α` or `m β`, and `a` or `b` for data of type α or β.

Monads are an abstract concept with many applications. The option type is only one instance among many. The following table gives an overview of some monad instances and their effects.

Type	Effect
`id`	no effects
`Option`	simple exceptions
`fun α ↦ σ → α × σ`	threading through a state of type σ
`Set`	nondeterministic computations returning α values
`fun α ↦ t → α`	reading elements of type `t` (e.g., a configuration)
`fun α ↦ ℕ × α`	adjoining running time (e.g., to model time complexity)
`fun α ↦ String × α`	adjoining text output (e.g., for logging)
`IO`	interaction with the operating system
`TacticM`	interaction with the proof assistant

All of the above are unary type constructors `m : Type → Type`. Some effects can be combined (e.g., `fun α ↦ Option (t → α)`). Some effects are not executable (e.g., `Set`); these are nonetheless useful for modeling programs abstractly. Specific type constructors `m` may provide further operators beyond `pure` and `bind`. For example, they may provide a way to extract the boxed value.

Monads have several benefits. They provide the highly readable `do` notation. They support generic operations, such as `List.mmap {α β : Type} : (α → m β) → List α → m (List β)`, that work uniformly for all monads `m`. To quote *Programming in Lean* [2]:

> The power of the abstraction is not only that it provides general functions and notation that can be used in all these various instantiations, but also that it provides a helpful way of thinking about what they all have in common.

Besides being a useful computer science concept, monads provide a nice example of axiomatic reasoning.

The `bind` and `pure` operations are normally required to obey three laws. The `bind` operation combines two programs. If either of these is a pure program, we can inline it and eliminate the `bind`. This gives us the first two laws:

```
do                                  do
  let a' ← pure a      =            f a
  f a'
```

and

```
do                                  do
  let a ← ma           =            ma
  pure a
```

The third law is an associativity rule for bind. It allows us to flatten a nested computation:

```
do                                    do
  let b ←                               let a ← ma
    do                          =         let b ← f a
      let a ← ma                          g b
      f a
  g b
```

Earlier we compared a monad to a box. It may help to think of the box more concretely as a Swiss bank account, with `α :=` money. The first law means that if you put some money in the account, you can take it out. The second law means that nobody will notice if you take out some money and put it back afterwards. The third law means that performing two bank operations together followed by a third one is the same as performing the first one alone followed by the other two. Given the Swiss banks' reputation for secrecy, all three laws seem plausible.

7.3 A Type Class

Monads are a mathematical structure, so we use a type class to specify them in Lean. Recall that a type class is a structure type that is parameterized, typically by a type, but here by a type constructor `m : Type → Type`. Whenever we use a field from the type class on a concrete `m`, the type class inference mechanism retrieves the relevant structure value—the type class instance.

A possible Lean definition of monads, together with the three laws, follows:

```
class LawfulMonad (m : Type → Type)
  extends Pure m, Bind m where
  pure_bind {α β : Type} (a : α) (f : α → m β) :
    (pure a >>= f) = f a
  bind_pure {α : Type} (ma : m α) :
    (ma >>= pure) = ma
  bind_assoc {α β γ : Type} (f : α → m β) (g : β → m γ)
      (ma : m α) :
    ((ma >>= f) >>= g) = (ma >>= (fun a ↦ f a >>= g))
```

Let us study this definition step by step:

- We are creating a structure parameterized by a unary type constructor `m`—i.e., a value of type `Type → Type`.
- The structure inherits the fields, and any syntactic sugar, from structures called `Pure` and `Bind`. These provide the `pure` and `bind` operations on `m`, with the expected types and the syntactic sugar.
- Finally, three fields (`pure_bind`, `bind_pure`, and `bind_assoc`) are added to those already provided by `Pure` and `Bind`. Each field is a proof of one of the three laws.

We call our type class `LawfulMonad` because the three laws are required to hold. To instantiate this definition, we must supply the type constructor `m`, suitable `bind` and `pure` operators, and proofs of the laws.

Lean includes its own concept of monads, also called `LawfulMonad`. It is roughly equivalent to our definition but is distributed over multiple type classes.

7.4 No Effects

Lean's constant `id {α : Type} : α → α` is defined as the identity function `fun x ↦ x`. The identity type constructor is obtained by taking `α := Type`. We can register it as a monad:

```
def id.pure {α : Type} : α → id α
  | a => a

def id.bind {α β : Type} : id α → (α → id β) → id β
  | a, f => f a

instance id.LawfulMonad : LawfulMonad id :=
  { pure       := id.pure
    bind       := id.bind
    pure_bind  :=
      by
        intro α β a f
        rfl
    bind_pure  :=
      by
        intro α ma
        rfl
    bind_assoc :=
      by
        intro α β γ f g ma
        rfl }
```

The registration process requires us to provide five components: the `pure` and `bind` operations and the proofs of the three laws.

The *identity monad* is the simplest monad possible. It provides a simple box, with a single value in it, without any effect. It plays a similar role as 0 in additive arithmetic. We can think of the other monads as variations of it; for example, the option monad is an identity monad enriched with a special `Option.none` value representing an error state.

7.5 Basic Exceptions

As we saw above, the option type provides a basic exception mechanism. The following code shows how to register `Option : Type → Type` as a lawful monad:

```
def Option.pure {α : Type} : α → Option α :=
  Option.some

def Option.bind {α β : Type} :
  Option α → (α → Option β) → Option β
  | Option.none,   _ => Option.none
  | Option.some a, f => f a

instance Option.LawfulMonad : LawfulMonad Option :=
```

```
  { pure       := Option.pure
    bind       := Option.bind
    pure_bind  :=
      by
        intro α β a f
        rfl
    bind_pure  :=
      by
        intro α ma
        cases ma with
        | none   => rfl
        | some _ => rfl
    bind_assoc :=
      by
        intro α β γ f g ma
        cases ma with
        | none   => rfl
        | some _ => rfl }
```

The three proofs are straightforward.

Beyond the standard operations, it can be useful to throw and catch exceptions. This can be implemented as follows:

```
def Option.throw {α : Type} : Option α :=
  Option.none

def Option.catch {α : Type} : Option α → Option α → Option α
  | Option.none,   ma' => ma'
  | Option.some a, _   => Option.some a
```

The `Option.throw` operation raises an exception, leaving the program in an error state (`Option.none`). The `Option.catch` operation can be used to recover from an earlier exception. If the program is currently in an error state, `Option.catch` invokes some exception-handling code (its second argument). This code might in turn raise a new exception. If `Option.catch` is applied to a normal state (of the form `Option.some a`), nothing happens.

As a convenient alternative to `Option.catch ma ma'`, Lean supports the syntax `ma.catch ma'`. Here is a schematic example that demonstrates throwing and catching with this syntax:

```
do
  ...
  if ... then
    Option.throw
  else
    ...
.catch do
  ...
```

The corresponding Java code would look as follows:

```
try {
```

```
        ...
        if (...) {
            throw new UnknownException();
        } else {
            ...
        }
    } catch (UnknownException e) {
        ...
    }
```

Options cater for only one kind of error state. A more general abstraction, called *error monad*, supports different errors, as with the exceptions of Java and other programming languages.

7.6 Mutable State

The *state monad* provides an abstraction corresponding to a mutable state. For some programming languages, the compiler can detect the use of the state monad and translate programs using them to more efficient imperative programs.

Admittedly, "abstraction corresponding to a mutable state" may sound somewhat abstract, so let us consider a semiconcrete example. If you have some experience with functional programming, you probably at some point or other have written code that looks very much like this fragment:

```
def welcomeNewUser (userName : String) (ctxt : Context) :
  (ℕ × String) × Context :=
  let
    (user, ctxt') := createUser userName ctxt
    (password, ctxt'') := generateTemporaryPassword user ctxt'
    (ok, ctxt''') := sendUnencryptedEmail user password ctxt''
  in
    ((user, password), ctxt''')
```

(It is very important to send the password in an unencrypted email, and to ignore the `ok` status of the function.) The function takes some global state or context as input and invokes three functions in turn, each of which takes a context value and returns both some data and a new context. The context is effectively *threaded through* the program. This allows us to have a mutable state in a programming language without side effects.

The above approach is error-prone—it is all too easy to forget one prime (') and pass the wrong context to a function. The code is also cluttered by all the context variables. What if we could simply write the following instead?

```
def welcomeNewUserDo (userName : String) :
  Context → (ℕ × String) × Context :=
  do
    let user ← createUser userName
    let password ← generateTemporaryPassword user
    let ok ← sendUnencryptedEmail user password
    pure (user, password)
```

This is exactly what the state monad provides.

The state monad builds on top of a binary type constructor `Action`, which captures the concept of computations or actions over states of type σ with return values of type α. In Lean, `Action σ α` is defined as equal to σ → α × σ:[1]

```
def Action (σ α : Type) : Type :=
  σ → α × σ
```

(Since types are terms, we can use `def` also to define type abbreviations.) For a given type σ, we have that `Action σ : Type → Type` is a monad. The type σ abstracts over the exact memory layout. We could use tuples or lists to represent memory, for example, and instantiate the abstract state σ accordingly.

A stateful action is a function that takes some state and returns both a value and some new state. The σ → part of `Action`'s definition gives the old state; the left component of the cartesian product, α, gives the result of the computation; and the right component of the product, σ, gives the new state. Thus, the state is implicitly threaded through the program. As with other effectful programs, the `do` notation only exposes the data—the value of type α—and conceals the effect—the old and new σ states.

The case where σ := `Unit`, a type of cardinality one (similar to `void` in C or Java) whose unique value is written `()`, corresponds to the identity monad: The type `Unit → α × Unit` is isomorphic to α. This intuition can guide us when defining `pure` and `bind`.

We start by defining basic operations: two operations, `read` and `write`, to access the memory, and the standard operations `bind` and `pure`:

```
def Action.read {σ : Type} : Action σ σ
  | s => (s, s)

def Action.write {σ : Type} (s : σ) : Action σ Unit
  | _ => ((), s)

def Action.pure {σ α : Type} (a : α) : Action σ α
  | s => (a, s)

def Action.bind {σ : Type} {α β : Type} (ma : Action σ α)
    (f : α → Action σ β) :
  Action σ β
  | s =>
    match ma s with
    | (a, s') => f a s'
```

The `read` operation simply returns the current state `s` (in the first pair component) and leaves the state unchanged (in the second pair component). The `write` operation replaces the current state with `s` and returns `()`. The `pure` operation returns the current state `s` unchanged tupled together with the given value `a`. The `bind` operation passes the initial state to the `ma` argument, yielding a result `a` and a new state `s'`. These are passed to `f`, which returns a new result and a new state.

To register the `Action` type constructor as a lawful monad, we need as before to prove the three laws:

[1] This type constructor is traditionally called `State`, but this name is very confusing.

```
instance Action.LawfulMonad {σ : Type} :
  LawfulMonad (Action σ) :=
  { pure       := Action.pure
    bind       := Action.bind
    pure_bind  :=
      by
        intro α β a f
        simp [Pure.pure, Bind.bind, Action.pure, Action.bind]
    bind_pure  :=
      by
        intro α ma
        simp [Pure.pure, Bind.bind, Action.pure, Action.bind]
    bind_assoc :=
      by
        intro α β γ f g ma
        simp [Pure.pure, Bind.bind, Action.pure, Action.bind] }
```

Notice that, in all three proofs, we unfold the definitions of the generic `pure` and `bind` constants to expand them to `Action.pure` and `Action.bind`, whose definitions we also expand.

As a concrete example, the following program removes all elements that are smaller than a previous element in the list, leaving us with a list of increasing elements. The maximal element is stored as the state σ. Notice how the state is accessed by `read` and `write`.

```
def increasingly : List ℕ → Action ℕ (List ℕ)
  | []        => pure []
  | (n :: ns) =>
    do
      let prev ← Action.read
      if n < prev then
        increasingly ns
      else
        do
          Action.write n
          let ns' ← increasingly ns
          pure (n :: ns')
```

To execute the program, we must supply an initial state. The last state is returned along with the resulting list. It corresponds to the largest element encountered in the list or the start state. Thus, the commands

```
#eval increasingly [1, 2, 3, 2] 0
#eval increasingly [1, 2, 3, 2, 4, 5, 2] 0
```

produce the output

```
([1, 2, 3], 3)
([1, 2, 3, 4, 5], 5)
```

7.7 Nondeterminism

Whereas the option monad stores zero or one α values and the identity and state monads store exactly one value, the *set monad* stores a possibly infinite number of values. This is useful to model nondeterminism, as a set of possible behaviors.

Lean's type `Set α` is defined as `α → Prop`. In other words, a set is identified with its characteristic predicate. Familiar operators such as the empty set (∅), the universal set (`Set.univ`), union (∪), intersection (∩), and membership (∈) are supported, as well as traditional curly brace notations such as `{a}`, `{a, b}`, and `{x | P x}`. Many set constructs can be simplified by `simp`.

The `Set` type constructor can be registered as a lawful monad as follows:

```
def Set.pure {α : Type} : α → Set α
  | a => {a}

def Set.bind {α β : Type} : Set α → (α → Set β) → Set β
  | A, f => {b | ∃a, a ∈ A ∧ b ∈ f a}

instance Set.LawfulMonad : LawfulMonad Set :=
  { pure       := Set.pure
    bind       := Set.bind
    pure_bind  :=
      by
        intro α β a f
        simp [Pure.pure, Bind.bind, Set.pure, Set.bind]
    bind_pure  :=
      by
        intro α ma
        simp [Pure.pure, Bind.bind, Set.pure, Set.bind]
    bind_assoc :=
      by
        intro α β γ f g ma
        simp [Pure.pure, Bind.bind, Set.pure, Set.bind]
        apply Set.ext
        aesop }
```

The `pure` operation simply puts the given value `a` in a singleton set `{a}`. The `bind` operation calls `f` on all values in the set `A` and returns the union of all the results. For example, if `A := {3, 8}` and `f := (fun a ↦ {a + 1, a + 2})`, then `set.bind A f` equals `{4, 5, 9, 10}`.

The last proof relies on *set extensionality*, which states that two sets that contain the same elements must be equal:

```
Set.ext {α : Type} {A B : set α} : (∀x, x ∈ A ↔ x ∈ B) → A = B
```

Another noteworthy point is the use of the `aesop` tactic. The goal's target is

```
∀x, x ∈ b | ∃ a, (∃ a_1, a_1 ∈ ma ∧ a ∈ f a_1) ∧ b ∈ g a
    ↔ x ∈ b | ∃ a, a ∈ ma ∧ ∃ a_1, a_1 ∈ f a ∧ b ∈ g a_1
```

where the two sides of ↔ are the same except for the placement of the existential quantifiers (and, confusingly, the names of the bound variables). This ugly propo-

sition could be proved by a tedious sequence of introduction and elimination, but we deserve more automation.

7.8 Aesop Tactic

aesop

The `aesop` tactic [19], whose name stands for Automated Extensible Search for Obvious Proofs, is a general-purpose proof search tactic. Among others, it performs elimination of the logical symbols $\wedge$, $\vee$, $\leftrightarrow$, and $\exists$ in hypotheses and introduction of $\wedge$, $\leftrightarrow$, and $\exists$ in the target, and it regularly invokes `simp`. It can succeed at proving a goal, fail, or succeed partially, leaving some unfinished subgoals to the user.

7.9 A Generic Algorithm: Iteration over a List

Suppose we apply an effectful function `f` on all elements of a list using `map`. We then obtain a regular list of effectful values. For example:

```
def nthsFine {α : Type} (xss : List (List α)) (n : ℕ) :
  List (Option α) :=
  List.map (fun xs ↦ nth xs n) xss
```

The function `nthsFine xss n` tries to extract the `(n + 1)`st element of each list in `xss`. Running

```
#eval nthsFine [[11, 12, 13, 14], [21, 22, 23]] 2
```

returns `[Option.some 13, Option.some 23]`.

These `Option.some` constructors can be inconvenient. Often, we only care about whether any error arose. This leads us to our final example: a generic effectful program `mmap` that iterates over a list and applies an effectful function `f` to each element. The definition is recursive:

```
def mmap {m : Type → Type} [LawfulMonad m] {α β : Type}
    (f : α → m β) :
  List α → m (List β)
  | []      => pure []
  | a :: as =>
    do
      let b ← f a
      let bs ← mmap f as
      pure (b :: bs)
```

Notice that the function returns a single `m` value containing a list and not a list of `m` values. Try to work out why it is well typed and has the desired behavior.

We can now trying using `mmap` instead of `map`:

```
def nthsCoarse {α : Type} (xss : List (List α)) (n : ℕ) :
  Option (List α) :=
  mmap (fun xs ↦ nth xs n) xss
```

Running

```
#eval nthsCoarse [[11, 12, 13, 14], [21, 22, 23]] 2
```

returns `Option.some [13, 23]`, with a single `Option.some` around a pure list.

The `mmap` function distributes over the append operator `++`. The `do` notation is useful not only for defining functions but also for stating their properties:

```
theorem mmap_append {m : Type → Type} [LawfulMonad m]
    {α β : Type} (f : α → m β) :
  ∀as as' : List α, mmap f (as ++ as') =
    do
      let bs ← mmap f as
      let bs' ← mmap f as'
      pure (bs ++ bs')
  | [],      _   =>
    by simp [mmap, LawfulMonad.bind_pure, LawfulMonad.pure_bind]
  | a :: as, as' =>
    by simp [mmap, mmap_append _ as as', LawfulMonad.pure_bind,
      LawfulMonad.bind_assoc]
```

7.10 Summary of New Lean Constructs

Notations

`do`	indicates the start of an effectful program
`let ... ← ...`	assigns a variable in an effectful program
`>>=`	composes effectful computations

Theorem

`Set.ext`	set extensionality

Tactic

`aesop`	proves propositions using a generic search procedure

Chapter 8

Metaprogramming

Like most proof assistants, Lean can be extended with custom tactics and other functionality. Programming Lean itself, as opposed to merely using it, is called metaprogramming. Lean's metaprogramming framework uses mostly the same notions and syntax as Lean's input language, so that we do not need to learn a different language to program Lean. Monads are used to access Lean's state.

Abstract syntax trees, presented as inductive types, reflect internal data structures. The proof assistant's internals are exposed through Lean functions, which we can use to access the current goal, unify terms, query and modify the global context, and set attributes (e.g., `@[simp]`).

Here are some example applications of metaprogramming:

- goal transformations (e.g., applying safe introduction rules, putting the goal in negation normal form);
- heuristic proof search (e.g., applying unsafe introduction rules with backtracking);
- decision procedures (e.g., for linear arithmetic, propositional logic);
- definition generators (e.g., Haskell-style `deriving` for inductive types);
- advisor tools (e.g., theorem finders, counterexample generators);
- exporters (e.g., documentation generators);
- ad hoc proof automation (to avoid boilerplate or duplication).

As mathematician and Lean user Kevin Buzzard wrote:[1]

> If you find yourself "grinding" (to use a computer game phrase), doing the same sort of stuff over and over again because you need to do it to make progress, then you can try to persuade a computer scientist to write a tactic to do it for you (or even write your own tactic if you're brave enough to write `meta` Lean code).

8.1 Tactic Combinators

First, some terminology: A tactic *fails* if applying it produces an error; otherwise, it *succeeds*. One way for a tactic to succeed is to fully prove the goal. Another is

[1] `https://xenaproject.wordpress.com/2020/02/09/lean-is-better-for-proper-maths-than-all-the-other-theorem-provers/`

to produce new subgoals that replace the current goal. Some tactics succeed by doing nothing.

When programming our own tactics, we often need to repeat some actions on several goals, or to recover if a tactic fails. Tactic combinators help in such cases. One of the most useful tactic combinators is `repeat' tactic`. It invokes *tactic* repeatedly on all goals, then on the emerging subgoals, then on the emerging subsubgoals, and so on until *tactic* fails on all the available goals. Here is an example of `repeat'` involving the `Even` predicate presented in Chapter 6:

```
theorem repeat'_example :
  Even 4 ∧ Even 7 ∧ Even 3 ∧ Even 0 :=
  by
    repeat' apply And.intro
    repeat' apply Even.add_two
```

After the first `repeat'` line, the proof state consists of four goals:

```
⊢ Even 4        ⊢ Even 7        ⊢ Even 3        ⊢ Even 0
```

Notice that all conjunctions are gone. The second `repeat'`, which applies the theorem `Even.add_two : ∀k, Even k → Even (k + 2)` over and over, leaves us with these goals:

```
⊢ Even 0        ⊢ Even 1        ⊢ Even 1        ⊢ Even 0
```

The first and last goals are annoying because they correspond to the theorem `Even.zero`. We can prove them by trying to apply `Even.zero` whenever applying `Even.add_two` fails. This is achieved as follows:

```
theorem repeat'_first_example :
  Even 4 ∧ Even 7 ∧ Even 3 ∧ Even 0 :=
  by
    repeat' apply And.intro
    repeat'
      first
      | apply Even.add_two
      | apply Even.zero
```

The tactic combinator `first` | $tactic_1$ | ⋯ | $tactic_n$ first tries to execute its first argument, $tactic_1$. If this fails, $tactic_2$ is tried, and so on. If all the specified tactics fail, the entire combinator fails. In the example above, we have two unprovable goals left:

```
⊢ Even 1                ⊢ Even 1
```

The next combinator, `all_goals` *tactic*, invokes a tactic exactly once on each goal. The combinator succeeds only if *tactic* succeeds on *all* goals. It fails in the example below, because `Even.add_two` cannot be applied to the goal `⊢ Even 0`:

```
theorem all_goals_example :
  Even 4 ∧ Even 7 ∧ Even 3 ∧ Even 0 :=
  by
    repeat' apply And.intro
    all_goals apply Even.add_two    -- fails
```

To ignore failures of *tactic*, we can wrap it in the `try` combinator:

```
theorem all_goals_try_example :
  Even 4 ∧ Even 7 ∧ Even 3 ∧ Even 0 :=
  by
    repeat' apply And.intro
    all_goals try apply Even.add_two
```

The resulting state is

```
⊢ Even 2        ⊢ Even 5        ⊢ Even 1        ⊢ Even 0
```

The construct `try` *tactic* is equivalent to `first` | *tactic* | `skip`, where `skip` is a tactic that succeeds without doing anything. Hence `try` *tactic* always succeeds. A related tactic is `done`: It succeeds if there are no goals left and fails otherwise.

Another variant is the `any_goals` *tactic* combinator. It tries to invoke *tactic* once on each goal, but unlike `all_goals`, it succeeds if *tactic* succeeds on *any* goal. The example

```
theorem any_goals_example :
  Even 4 ∧ Even 7 ∧ Even 3 ∧ Even 0 :=
  by
    repeat' apply And.intro
    any_goals apply Even.add_two
```

results in the state

```
⊢ Even 2        ⊢ Even 5        ⊢ Even 1        ⊢ Even 0
```

This is the same state as in the previous example. In general, the difference is that `any_goals` *tactic* can fail whereas `all_goals try` *tactic* always succeeds.

Sometimes we want to leave a goal alone unless we can fully prove it. The combinator `solve` | $tactic_1$ | ⋯ | $tactic_n$ first tries to execute its first argument, $tactic_1$. If this fails to prove the goal, $tactic_2$ is tried, and so on. If all the specified tactics fail to prove the goal, the entire combinator fails. (Compare this behavior with that of `first` | $tactic_1$ | ⋯ | $tactic_n$, which only requires one of the specified tactics to succeed, not to prove the goal.) Consider this example:

```
theorem any_goals_solve_repeat_first_example :
  Even 4 ∧ Even 7 ∧ Even 3 ∧ Even 0 :=
  by
    repeat' apply And.intro
    any_goals
      solve
      | repeat'
          first
          | apply Even.add_two
          | apply Even.zero
```

The first and fourth goals are proved, and we are left with the two unprovable goals exactly as they stand in the theorem statement:

```
⊢ Even 7                    ⊢ Even 3
```

Note that the `repeat'` combinator can lead to infinite looping. Consider this example:

```
theorem repeat'_Not_example :
  ¬ Even 1 :=
  by repeat' apply Not.intro
```

The `Not.intro` rule is `(?a → False) → ¬ ?a`, so it applies once to transform the goal into `⊢ Even 1 → False`. Because `¬ ?a` is defined as `?a → False`, the rule applies again, yielding the same goal again. The tactic loops.

Finally, the "and then" operator `<;>` can be used to join two tactics. The left-hand side is executed on the first goal. The right-hand side is executed on every emerging subgoal (but not on the original second goal, third goal, etc.). Thus, we can write

```
by
  induction n <;>
    aesop
```

instead of the more verbose

```
by
  induction n with
  | zero       => aesop
  | succ n' ih => aesop
```

8.2 Macros

It is time to start with the actual metaprogramming, by coding a custom tactic as a macro. The tactic embodies the behavior we hardcoded in the `solve` example above:

```
macro "intro_and_even" : tactic =>
  `(tactic|
      (repeat' apply And.intro
       any_goals
         solve
         | repeat'
             first
             | apply Even.add_two
             | apply Even.zero))
```

The first line declares `intro_and_even` as a macro belonging to the `tactic` syntactic category. On the remaining lines, the `` `(tactic| `tactic`) `` construct embeds the specified tactic, *`tactic`*, in a macro. The tactic itself is specified using standard syntax.

Once we have defined a custom tactic, we can invoke it in a proof:

```
theorem intro_and_even_example :
  Even 4 ∧ Even 7 ∧ Even 3 ∧ Even 0 :=
  by
    intro_and_even
```

This yields the subgoals

```
⊢ Even 7                    ⊢ Even 3
```

8.3 The Metaprogramming Monads

Macros are a simple mechanism that can be used to program simple proof automation. For most metaprogramming tasks, however, we need to use the metaprogramming monads, `MetaM` and `TacticM`.

`MetaM` combines the attributes of several kinds of monads:

- It is a state monad providing access to the global context (including all definitions and inductive types), notations, and attributes (e.g., the list of `@[simp]` theorems), among others.
- It behaves like an option monad. The metaprogram `failure` signals that a tactic has failed.
- It supports tracing, so we can use the program `logInfo` to display messages.
- Like other monads, it supports imperative constructs such as `for–in` loops, `continue` statements, and `return` statements.

`TacticM` extends `MetaM` with goal management: It provides access to the list of goals. It also allows us to run Lean to fill in the implicit `{ }` and type class `[ ]` arguments in expressions, to expand macros, and more.

Inside Lean, each goal is represented as a metavariable `?m` standing for a missing term (typically, a proof term). Each metavariable has a type (typically, a proposition) and a local context specifying the variables and hypotheses that can be used to prove the goal associated with the metavariable.

Let us put the metaprogramming monad to some use by defining a tactic that uses `TacticM`'s tracing facilities to display the Lean version number, the list of goals, and the target of the first goal (which Lean calls the *main* goal):

```
def traceGoals : TacticM Unit :=
  do
    logInfo m!"Lean version {Lean.versionString}"
    logInfo "All goals:"
    let goals ← getUnsolvedGoals
    logInfo m!"{goals}"
    match goals with
    | []     => return
    | _ :: _ =>
      logInfo "First goal's target:"
      let target ← getMainTarget
      logInfo m!"{target}"

elab "trace_goals" : tactic =>
  traceGoals
```

The code has many interesting features:

- The first line declares a function `traceGoals` of type `TacticM Unit`—the type of tactics that return `Unit` (a trivial type of cardinality one). Notice that a metaprogramming function is defined using the same syntax as any Lean function.
- The second line enters the monad. The remaining lines are effectful operations that access Lean's internals.

- The `m!"..."` syntax specifies a string in which every occurrence of {*term*} , where *term* is a Lean term, are evaluated and serialized into a string. For example, if `Lean.versionString` is "4.0.0-nightly-2023-08-19", then `m!"Lean version {Lean.versionString}"` evaluates to the string "Lean version 4.0.0-nightly-2023-08-19".
- The last two lines register a new tactic called `trace_goal` that simply calls the `traceGoals` function.

Here is an example of how we can use the new tactic:

```
theorem Even_18_and_Even_20 (α : Type) (a : α) :
  Even 18 ∧ Even 20 :=
  by
    apply And.intro
    trace_goals
    intro_and_even
```

The output, which is made visible by hovering over `trace_goals`, is as follows:

```
Lean version 4.0.0-nightly-2023-08-19
All goals:
[case left
α : Type
a : α
⊢ Even 18,
 case right
α : Type
a : α
⊢ Even 20]
First goal's target:
Even 18
```

Although Lean displays goals using the familiar goal syntax `C ⊢ P`, they are actually metavariables.

The constants used in the above program have the following types:

```
logInfo : MessageData → TacticM Unit
getUnsolvedGoals : TacticM (List MVarId)
getMainTarget : TacticM Expr
```

where `MessageData` represents a message, `MVarId` represents a metavariable identifier, and `Expr` represents a term.

8.4 First Example: An Assumption Tactic

Our first larger example implements a `hypothesis` tactic that, like the predefined `assumption` tactic, looks for a hypothesis of the right type (i.e., the right proposition) and applies it to prove the goal:

```
def hypothesis : TacticM Unit :=
  withMainContext
    (do
```

```
        let target ← getMainTarget
        let lctx ← getLCtx
        for ldecl in lctx do
          if ! LocalDecl.isImplementationDetail ldecl then
            let eq ← isDefEq (LocalDecl.type ldecl) target
            if eq then
              let goal ← getMainGoal
              MVarId.assign goal (LocalDecl.toExpr ldecl)
              return
        failure)

elab "hypothesis" : tactic =>
  hypothesis
```

In the `hypothesis` function, we first extract the first goal's target and the local context. To make sure that `getLCtx` gives us the local context of the current first goal, we pass the entire `do` block to the `withMainContext` function. In general, any `TacticM` computation is performed within an ambient local context that gives meaning to the free variables occurring in expressions. The `withMainContext` function sets this local context to the local context of the current first goal.

Inside the `do` block, we iterate over all declarations in the local context, using the convenient monadic construct `for-in`. For each local variable or hypothesis `h` that is not a so-called implementation detail (i.e., a hypothesis inserted by Lean that is invisible to users), we check if its type (typically, its proposition) is equal to the target up to computation and instantiation of metavariables, and if so, we get the metavariable `?m` associated with the first goal and assign `?m := h`, thereby proving the goal. Finally, we `return`.

Since goals are representing by metavariables, assigning a term to a metavariable `?m` is Lean's low-level way of proving a goal. New metavariables occurring in that term correspond to new subgoals that must be proved.

A simple invocation of `hypothesis` follows:

```
theorem hypothesis_example {α : Type} {p : α → Prop} {a : α}
    (hpa : p a) :
  p a :=
  by hypothesis
```

If we add tracing, we can see that `α`, `p`, `a`, and `hpa` are tried in turn before the matching hypothesis `hpa` is found and successfully applied.

The example used the following new constants:

```
                          getLCtx : TacticM LocalContext
LocalDecl.isImplementationDetail : LocalDecl → Bool
                          isDefEq : Expr → Expr → TacticM Bool
                   LocalDecl.type : LocalDecl → Expr
                      getMainGoal : TacticM MVarId
                     MVarId.assign : MVarId → Expr → TacticM Unit
                   LocalDecl.toExpr : LocalDecl → Expr
                  failure {α : Type} : TacticM α
```

8.5 Expressions

The metaprogramming framework revolves around the type `Expr` of expressions or terms. An important component of expressions are names, of type `Name`. We start with names.

Names can be specified using a single backtick. For example, `` `x `` represents the name `x`, which can be given to a variable or constant. When referring to constants, we must specify the full name, including namespaces; thus, to refer to the `Even` predicate of Chapter 6, we must write `` `LoVe.Even `` and not `` `Even ``.

If we want to refer to an existing constant, Lean offers the double-backtick syntax, which looks up the name using Lean's usual name elaboration rules and expands it to its full name. Thus, both `` ``Even `` and `` ``LoVe.Even `` refer to the name `LoVe.Even`, and if we write some name that is not declared, such as `` ``EvenIf ``, Lean gives an error.

The type `Expr` is defined as follows:

```
inductive Expr : Type where
  | const   : Name → List Level → Expr
  | sort    : Level → Expr
  | fvar    : FVarId → Expr
  | mvar    : MVarId → Expr
  | app     : Expr → Expr → Expr
  | lam     : Name → Expr → Expr → BinderInfo → Expr
  | bvar    : Nat → Expr
  | forallE : Name → Expr → Expr → BinderInfo → Expr
  | letE    : Name → Expr → Expr → Expr → Bool → Expr
  | lit     : Literal → Expr
  | mdata   : MData → Expr → Expr
  | proj    : Name → Nat → Expr → Expr
```

Let us review the main constructors:

- `Expr.const name levels` represents a constant called `name`, such as `Nat.add` or $\mathbb{N}$. The `levels` argument represents universe levels, a concept that will be explained in Chapter 12. For example, `Expr.const ``Nat.add []` represents `Nat.add` and `Expr.const ``Nat []` represents `Nat` (i.e., $\mathbb{N}$).
- `Expr.sort level` is used to represent the types of types. For example, `Expr.sort Level.zero` stands for `Prop`, and `Expr.sort (Level.succ Level.zero)` stands for `Type`.
- `Expr.fvar id` represents a free variable in the local context (e.g., `a`, `h`). The `id` argument is a unique identifier for the variable.
- `Expr.mvar id` represents a metavariable, i.e., a variable `?m` with a question mark. The `id` argument is a unique identifier for the metavariable.
- `Expr.app t u` represents the application of a function `t` to an argument `u`. For example, `Expr.app (Expr.const ``Nat.succ []) (Expr.const ``Nat.zero [])` represents `Nat.succ Nat.zero`.
- `Expr.lam name σ t bi` represents an anonymous function (or λ-expression). The `name` argument is the bound variable's name, the `σ` argument is the bound variable's type, the `t` argument is the function's body, and the `bi`

argument stores whether the variable is an explicit `( )`, implicit `{ }`, or type class `[ ]` argument.

- `Expr.bvar i` represents a bound variable, using a notation known as De Bruijn index. `Expr.var 0` refers to the variable bound by the closest binder, `Expr.var 1` refers to the variable bound by the second closest binder, and so on. Thus,

```
Expr.lam `x (Expr.const ``Nat []) (Expr.bvar 0)
  BinderInfo.default
```

  represents `fun x : ℕ ↦ x`, and

```
Expr.lam `x (Expr.const ``Nat [])
  (Expr.lam `y (Expr.const ``Nat []) (Expr.bvar 1)
     BinderInfo.default)
  BinderInfo.default
```

  represents `fun x y : ℕ ↦ x`.
- `Expr.forallE name σ τ bi` represents a possibly dependent function type. The `name` argument is the name of the bound variable, the `σ` argument is the domain type, the `τ` argument is the result type, and `bi` is as for `Expr.lam` above. For example,

```
Expr.forallE `n (Expr.const ``Nat [])
  (Expr.app (Expr.const ``Even []) (Expr.bvar 0))
  BinderInfo.default
```

  represents `(n : ℕ) → Even n` (also written `∀n : ℕ, Even n`), and

```
Expr.forallE `dummy (Expr.const `Nat [])
  (Expr.const `Bool []) BinderInfo.default
```

  represents `ℕ → Bool`.

8.6 Second Example: A Conjunction-Destructing Tactic

In this and the next section, we define two further tactics that accomplish well-defined tasks. The first of these two tactics, called `destruct_and`, automates the elimination of conjunctions in premises. Our aim is to automate proofs such as the following:

```
theorem abc_a (a b c : Prop) (h : a ∧ b ∧ c) :
  a :=
  And.left h

theorem abc_b (a b c : Prop) (h : a ∧ b ∧ c) :
  b :=
  And.left (And.right h)

theorem abc_bc (a b c : Prop) (h : a ∧ b ∧ c) :
  b ∧ c :=
```

```
  And.right h

theorem abc_c (a b c : Prop) (h : a ∧ b ∧ c) :
  c :=
  And.right (And.right h)
```

In each case, we would like to simply write `by destruct_and h` as the proof.

Our tactic relies on a helper function, which takes as argument a proof term `hP` (initially the hypothesis `h`) from which we extract conjuncts:

```
partial def destructAndExpr (hP : Expr) : TacticM Bool :=
  withMainContext
    (do
       let target ← getMainTarget
       let P ← inferType hP
       let eq ← isDefEq P target
       if eq then
         let goal ← getMainGoal
         MVarId.assign goal hP
         return true
       else
         match Expr.and? P with
         | Option.none        => return false
         | Option.some (Q, R) =>
           let hQ ← mkAppM ``And.left #[hP]
           let success ← destructAndExpr hQ
           if success then
             return true
           else
             let hR ← mkAppM ``And.right #[hP]
             destructAndExpr hR)
```

Like in `hypothesis`, we pass the entire `do` block to the `withMainContext` function. This ensures that `inferType` and `isDefEq` operate within the right local context. Inside the `do` block, we first extract the first goal's target and `hP`'s type (typically, its proposition) `P`. If they are equal up to computation and instantiation of metavariables, we close the goal by assigning to its metavariable, as we did in the `hypothesis` example, and we return `true` to indicate success. Otherwise, we check whether `hP`'s proposition is of the form `Q ∧ R`. If so, we call the helper function recursively with the proof term `hQ := And.left hP`, which is a proof of `Q`. If this succeeds, we are done; otherwise, we try the proof term `hR := And.right hP`, which is a proof of `R`.

Note the presence of the keyword `partial` at the front of the function definition. It is needed here because Lean failed to prove that the function always terminates. Since the function will only be used as a metaprogram, and not inside a proposition, termination is optional, and we can disable the termination check by specifying `partial`.

Also noteworthy is the `mkAppM` function, which is used to construct a curried application of a constant to an array of arguments. Arrays are similar to lists but are written with a prefixed `#` (e.g., `#[1, 2, 3]`). Using `mkAppM` is more convenient

than applying the `Expr.app` constructor multiple times. Additionally, `mkAppM` allows us to omit implicit arguments such as the propositions `Q` and `R`, which we would otherwise have to provide as arguments to `And.left` and `And.right`.

The main function has very little to do:

```
def destructAnd (name : Name) : TacticM Unit :=
  withMainContext
    (do
       let h ← getFVarFromUserName name
       let success ← destructAndExpr h
       if ! success then
         failure)

elab "destruct_and" h:ident : tactic =>
  destructAnd (getId h)
```

The function retrieves the hypothesis `h` using `getFVarFromUserName` and calls the helper function. If the helper returns `false`, the tactic fails.

We can now use our new widget on the motivating examples:

```
theorem abc_a_again (a b c : Prop) (h : a ∧ b ∧ c) :
  a :=
  by destruct_and h

theorem abc_b_again (a b c : Prop) (h : a ∧ b ∧ c) :
  b :=
  by destruct_and h

theorem abc_bc_again (a b c : Prop) (h : a ∧ b ∧ c) :
  b ∧ c :=
  by destruct_and h

theorem abc_c_again (a b c : Prop) (h : a ∧ b ∧ c) :
  c :=
  by destruct_and h
```

The following new constants were used in the above metaprogram:

```
              inferType : Expr → TacticM Expr
              Expr.and? : Expr → Option (Expr × Expr)
                 mkAppM : Name → Array Expr → TacticM Expr
  getFVarFromUserName : Name → TacticM Expr
```

8.7 Third Example: A Direct Proof Finder

Sometimes we state a theorem, prove it, and later realize that the theorem already exists. This can be prevented by using `prove_direct`, a tactic that traverses all available theorems and checks whether one of them can prove the current goal. We will review its code in steps.

The first step is a function `isTheorem` that returns `true` if a declaration is an axiom or a theorem and `false` otherwise:

```
def isTheorem : ConstantInfo → Bool
  | ConstantInfo.axiomInfo _ => true
  | ConstantInfo.thmInfo _   => true
  | _                        => false
```

We will use this function to filter out the declarations that do not interest us.

The next function applies the theorem named `name` to the current goal:

```
def applyConstant (name : Name) : TacticM Unit :=
  do
    let cst ← mkConstWithFreshMVarLevels name
    liftMetaTactic (fun goal ↦ MVarId.apply goal cst)
```

Given a name, the `mkConstWithFreshMVarLevels` function creates an expression `cst` representing the constant. The allusion to "fresh metavariable levels" in the function's name will become clearer in Chapter 12. `MVarId.apply` (not to be confused with `MVarId.assign`) then applies the constant to the current goal, setting `?m` := `cst` $\texttt{?m}_1 \ldots \texttt{?m}_n$ and returning the fresh metavariables $\texttt{?m}_j$, which represent the premises of `cst`.

The `liftMetaTactic` function retrieves the identifier of the first goal, runs the given function on the goal within the lower-level `MetaM` monad, and replaces the goal with the subgoals returned by the function.

The next function implements a combinator that behaves like `<;>` but that can be used from a metaprogram:

```
def andThenOnSubgoals (tac₁ tac₂ : TacticM Unit) :
  TacticM Unit :=
  do
    let origGoals ← getGoals
    let mainGoal ← getMainGoal
    setGoals [mainGoal]
    tac₁
    let subgoals₁ ← getUnsolvedGoals
    let mut newGoals := []
    for subgoal in subgoals₁ do
      let assigned ← MVarId.isAssigned subgoal
      if ! assigned then
        setGoals [subgoal]
        tac₂
        let subgoals₂ ← getUnsolvedGoals
        newGoals := newGoals ++ subgoals₂
    setGoals (newGoals ++ List.tail origGoals)
```

The `TacticM` monad keeps track of the current goals to prove. We can retrieve the list using `getGoals` and set it using `setGoals`. Setting the list of subgoals is useful if we want the tactic to temporarily focus on specific subgoals.

Here, we first focus on the first goal (`setGoals [mainGoal]`) and invoke the first tactic. For each subgoal that emerges, we focus on it (`setGoals [subgoal]`) and invoke the second tactic. All the unsolved subsubgoals emerging from the second tactic are collected in the mutable variable `newGoals`. Since proving a goal can sometimes instantiate another metavariable, we check at each iteration whether the current subgoal metavariable is assigned and skip the subgoal if it is.

At the end, we update the goals to include all the pending goals: those in `newGoals` and all but the first goal of `origGoals`, which we have not considered.

In general, at the end of a tactic, we should make sure that the list of goals consists of all goals that remain to be proved. Otherwise, we may get cryptic errors such as

```
declaration has metavariables
```

We also need a tactic that attempts to prove the goal using a theorem specified by its name and that invokes `hypothesis` to prove any emerging subgoals:

```
def proveUsingTheorem (name : Name) : TacticM Unit :=
  andThenOnSubgoals (applyConstant name) hypothesis
```

This is the programmatic equivalent of the proof `apply` *name* `<;> hypothesis`.

Finally, we are ready to review the main function:

```
def proveDirect : TacticM Unit :=
  do
    let origGoals ← getUnsolvedGoals
    let goal ← getMainGoal
    setGoals [goal]
    let env ← getEnv
    for (name, info)
        in SMap.toList (Environment.constants env) do
      if isTheorem info && ! ConstantInfo.isUnsafe info then
        try
          proveUsingTheorem name
          logInfo m!"Proved directly by {name}"
          setGoals (List.tail origGoals)
          return
        catch _ =>
          continue
    failure

elab "prove_direct" : tactic =>
  proveDirect
```

We focus on the first goal, then iterate over all constants declared in the environment. If the constant is a theorem and is not so-called unsafe, we try to apply it using our helper `proveUsingTheorem`. If this succeeds, we print "Proved directly by *name*," where *name* is the name of the theorem, and return. On failure, we keep iterating. If the entire iteration is exhausted, we report a failure.

Here is the tactic in action:

```
theorem Nat.symm (x y : ℕ) (h : x = y) :
  y = x :=
  by prove_direct
```

This prints "Proved directly by `symm`." The message is helpful because we can apply the specified theorem directly instead of relying on the relatively slow `prove_direct` tactic. Specifically, we can apply the theorem `symm` in conjunction with `hypothesis` as follows:

```
theorem Nat.symm_manual (x y : ℕ) (h : x = y) :
  y = x :=
  by
    apply symm
    hypothesis
```

Here is a list of new constants featured in this example:

```
mkConstWithFreshMVarLevels : Name → TacticM Expr
            liftMetaTactic : (MVarId → MetaM (List MVarId)) →
                             TacticM Unit
              MVarId.apply : MVarId → Expr → MetaM (List MVarId)
                  getGoals : TacticM (List MVarId)
                  setGoals : List MVarId → TacticM Unit
         MVarId.isAssigned : MVarId → TacticM Bool
                    getEnv : TacticM Environment
               SMap.toList : ConstMap → List (Name × ConstantInfo)
     Environment.constants : Environment → ConstMap
     ConstantInfo.isUnsafe : ConstantInfo → Bool
```

This concludes our review of `prove_direct`. A similar, more sophisticated tactic is available as `apply?` in `mathlib`.

8.8 Miscellaneous Tactics

Although the focus of this chapter was on developing new tactics, we encountered three predefined tactics.

skip

The `skip` tactic succeeds without doing anything. It is sometimes useful as a building block when we develop custom tactics.

done

The `done` tactic raises a failure if there are some goals left; otherwise, it succeeds without doing anything. Like `skip`, it can be useful as a building block.

apply?

The `apply?` tactic searches the loaded libraries for a lemma that exactly proves the goal. On success, it suggests a tactic invocation of the form `exact ...`, which can be inserted in the formalization.

8.9 Summary of New Lean Constructs

Declaration

`partial`	prefixes declarations of possibly nonterminating metaprograms

Quotations

`` `n ``	quotes a literal name
`` ``n ``	quotes a literal name with elaboration and checking

Tactics

`apply?`	searches for a theorem that proves the current goal
`done`	fails if there are some goals left
`skip`	does nothing

Tactic Combinators

`<;>`	invokes the second tactic on all subgoals from the first tactic
`all_goals`	invokes a tactic once on each goal, expecting only successes
`any_goals`	invokes a tactic once on each goal, expecting at least one success
`first | ... | ...`	tries the tactics in turn until one succeeds
`repeat'`	repeatedly invokes a tactic on all goals and subgoals until failure
`solve | ... | ...`	tries to fully prove the current goal using the tactics in turn
`try`	tries to invoke a tactic; does nothing on failure

Part III

Program Semantics

Chapter 9

Operational Semantics

In this and the next two chapters, we will see how to use Lean to specify the syntax and semantics of programming languages, to prove properties of the semantics, and to reason about concrete programs.

This chapter is heavily inspired by Chapter 7 of *Concrete Semantics: With Isabelle/HOL* [23].

9.1 Formal Semantics

A formal semantics allows us to specify and reason about a programming language and about individual programs written in that language. It can form the basis of verified compilers, interpreters, verifiers, static analyzers, type checkers, and more. Without formal proofs, these tools are almost always wrong.

Consider WebAssembly, a new machine-like language for web browsers, designed as a portable target for compiling high-level languages such as C++ and Rust. A researcher, Conrad Watt [29], formalized its semantics and type system using the Isabelle/HOL proof assistant. He found many issues (our italics):

> We have produced a full Isabelle mechanisation of the core execution semantics and type system of the WebAssembly language. In addition, we have created a mechanised proof for the type soundness properties stated in the working group's paper. In order to complete this proof, *several deficiencies* in the official WebAssembly specification, uncovered by our proof and modelling work, needed to be corrected by the specification authors. In some cases, these meant that *the type system was originally unsound*.
>
> We have maintained a constructive dialogue with elements of the working group, mechanising and verifying new features as they are added to the specification. In particular, the mechanism by which a WebAssembly implementation interfaces with its host environment was not formally specified in the working group's original paper. Extending our mechanisation to model this feature *revealed a deficiency* in the WebAssembly specification that *sabotaged the soundness of the type system*.

Watt's research is only one example among many. Proof assistants are widely used for programming language research. Every year, around 10%–20% of papers

presented at the Principles of Programming Languages (POPL) conference are formalized. This is possible because comparatively little machinery is needed to get started. The proofs tend to have lots of cases, which is a good match for computers. Moreover, proof assistants are extremely convenient to keep track of what needs to be changed as we extend a programming language with more features.

9.2 A Minimalistic Imperative Language

WHILE[1] is a minimalistic imperative language with the following grammar:

S ::=	`skip`	(no-op)
	x `:=` a	(assignment)
	S; S	(sequential composition)
	`if` b `then` S `else` S	(conditional statement)
	`while` b `do` S	(`while` loop)

where S stands for a *statement* (also called *command* or *program*), x for a program variable, a for an arithmetic expression, and b for a Boolean expression.

In our grammar, we deliberately leave the syntax of arithmetic and Boolean expressions unspecified. In Lean, we have the choice:

- We can use a type such as `AExp` from Section 2.1 and similarly for Booleans.
- We can simply decide that an arithmetic expression is a function from states to numbers (e.g., `State` $\to \mathbb{N}$) and a Boolean expression is a predicate over states (e.g., `State` $\to$ `Bool` or `State` $\to$ `Prop`). A *State* is a mapping from program variables to values. Thus, `x + y + 1` would be represented by the function `fun s : State` $\mapsto$ `s "x" + s "y" + 1`, and `a` $\neq$ `b` would be represented by the predicate `fun s : State` $\mapsto$ `s "a"` $\neq$ `s "b"`.

These two options correspond to the difference between deep and shallow embeddings. A *deep embedding* of some syntax (expression, formula, program, etc.) consists of an abstract syntax tree specified in the proof assistant (e.g., `AExp`) with a semantics (e.g., `eval`). In contrast, a *shallow embedding* simply reuses the corresponding mechanisms from the logic (e.g., functions and predicates).

A deep embedding allows us to reason about a program's syntax. A shallow embedding is more lightweight, because we can use it directly, without having to define a semantics. A shallow embedding is its own semantics.

In Chapter 7, we used a shallow embedding of effectful programs. Here, we will use a deep embedding of programs (which we find interesting and want to study closely) and a shallow embedding of arithmetic and Boolean expressions (which we find less interesting). Our Lean definition of programs follows:

```
inductive Stmt : Type where
  | skip       : Stmt
  | assign     : String → (State → ℕ) → Stmt
  | seq        : Stmt → Stmt → Stmt
  | ifThenElse : (State → Prop) → Stmt → Stmt → Stmt
  | whileDo    : (State → Prop) → Stmt → Stmt
```

[1] Fans of backronyms might enjoy this one: Weak Hypothetical Imperative Language Example.

The infix syntax `S ; T` abbreviates `Stmt.seq S T`.

The correspondence between the inductive type's constructors and the WHILE grammar rules should be clear. Variables are represented by strings. The type `State` is defined as `String → ℕ`, a mapping from variable names to values. For simplicity, our program variables are all of type natural number, and all possible variable names exist in the state and are assigned a value.

The following small program illustrates deep embedding:

```
def sillyLoop : Stmt :=
  Stmt.whileDo (fun s ↦ s "x" > s "y")
    (Stmt.skip;
     Stmt.assign "x" (fun s ↦ s "x" - 1))
```

9.3 Big-Step Semantics

An *operational semantics* corresponds to an idealized interpreter. There are two main variants: big-step semantics and small-step semantics. We will start by giving a big-step semantics to our WHILE language.

In a *big-step operational semantics* (also called *natural semantics*), judgments have the form $(S, s) \Longrightarrow t$ and the following intuitive interpretation:

> Starting in a state s, executing S may terminate in the state t.

For deterministic languages, since programs always have a single outcome, "may terminate" means the same as "must terminate" or "terminates."

In accordance with the definition of WHILE programs, a state s is a function of type $\texttt{String} \to \mathbb{N}$. An example judgment follows:

$$(x := x + y;\ y := 0,\ [x \mapsto 3,\ y \mapsto 5]) \Longrightarrow [x \mapsto 8,\ y \mapsto 0]$$

We use the informal notation $[x \mapsto 3,\ y \mapsto 5]$ to represent the function $\texttt{fun v} \mapsto \texttt{if v = "x" then 3 else if v = "y" then 5 else 0}$ and similarly for $[x \mapsto 8,\ y \mapsto 0]$. Intuitively, the judgment holds.

The traditional way to specify such a semantics is through a formal system of derivation rules, in the style of the typing rules presented in Sections 1.3 and 4.6. The derivation rules for big-step semantics judgments are given below. The rules can be seen as an idealized interpreter for WHILE programs.

$$\frac{}{(\texttt{skip}, s) \Longrightarrow s}\ \textsc{Skip}$$

$$\frac{}{(x := a, s) \Longrightarrow s[x \mapsto a\ s]}\ \textsc{Assign}$$

$$\frac{(S, s) \Longrightarrow t \qquad (T, t) \Longrightarrow u}{(S; T, s) \Longrightarrow u}\ \textsc{Seq}$$

$$\frac{(S, s) \Longrightarrow t}{(\texttt{if b then S else T}, s) \Longrightarrow t}\ \textsc{If-True} \quad \text{if } b\ s \text{ is true}$$

$$\frac{(T, s) \Longrightarrow t}{(\texttt{if b then S else T}, s) \Longrightarrow t}\ \textsc{If-False} \quad \text{if } b\ s \text{ is false}$$

$$\frac{(S, s) \Longrightarrow t \qquad (\texttt{while}\ b\ \texttt{do}\ S, t) \Longrightarrow u}{(\texttt{while}\ b\ \texttt{do}\ S, s) \Longrightarrow u}\ \textsc{While-True} \quad \text{if } b\ s \text{ is true}$$

$$\frac{}{(\texttt{while}\ b\ \texttt{do}\ S, s) \Longrightarrow s}\ \textsc{While-False} \quad \text{if } b\ s \text{ is false}$$

In the rules, `a s` denotes the value of arithmetic expression `a` in state `s`, and similarly for `b s`. Moreover, the syntax `s[x ↦ n]` represents the state that is identical to `s` except that it maps the variable `x` to `n`. Formally:

```
s[x ↦ n] = (fun v ↦ if v = x then n else s v)
```

This syntax is provided by `LoVelib`.

As an exercise, let us derive the example judgment above. Let `s := [x ↦ 3, y ↦ 5]`, `t := [x ↦ 8, y ↦ 5]`, and `u := [x ↦ 8, y ↦ 0]`. Then we have

$$\dfrac{\dfrac{}{(x := x + y, s) \Longrightarrow t}\ \textsc{Assign} \qquad \dfrac{}{(y := 0, t) \Longrightarrow u}\ \textsc{Assign}}{(x := x + y;\ y := 0, s) \Longrightarrow u}\ \textsc{Seq}$$

The derivation rules can be read intuitively. Consider Seq:

> If (1) executing `S` in state `s` leads to state `t` and (2) executing `T` in state `t` leads to state `u`, then executing the sequential composition `S; T` in state `s` leads to state `u`.

The conditions (1) and (2) correspond to the two premises of Seq.

The most complicated rule is undoubtedly While-True. Intuitively, it can be understood as follows:

> Assume condition `b` is true in state `s`. If (1) executing `S` in state `s` leads to state `t` and (2) executing `while b do S` from state `t` leads to state `u`, then executing `while b do S` in state `s` leads to state `u`.

Another way to think about While-True is in terms of loop unrolling. If the loop condition is true, `while b do S` is equivalent to the compound statement `S; while b do S`. The two premises of While-True correspond to the two premises of the instance of the Seq rule for `S; while b do S`.

In Lean, a big-step semantics judgment is represented by an inductive predicate whose introduction rules closely follow the derivation rules above:

```
inductive BigStep : Stmt × State → State → Prop where
  | skip (s) :
    BigStep (Stmt.skip, s) s
  | assign (x a s) :
    BigStep (Stmt.assign x a, s) (s[x ↦ a s])
  | seq (S T s t u) (hS : BigStep (S, s) t)
      (hT : BigStep (T, t) u) :
    BigStep (S; T, s) u
  | if_true (B S T s t) (hcond : B s)
      (hbody : BigStep (S, s) t) :
    BigStep (Stmt.ifThenElse B S T, s) t
```

```
  | if_false (B S T s t) (hcond : ¬ B s)
      (hbody : BigStep (T, s) t) :
    BigStep (Stmt.ifThenElse B S T, s) t
  | while_true (B S s t u) (hcond : B s)
      (hbody : BigStep (S, s) t)
      (hrest : BigStep (Stmt.whileDo B S, t) u) :
    BigStep (Stmt.whileDo B S, s) u
  | while_false (B S s) (hcond : ¬ B s) :
    BigStep (Stmt.whileDo B S, s) s
```

We use implicit arguments, within curly braces, for many of the variables corresponding to the derivation rule's mathematical variables.

Using an inductive predicate as opposed to a recursive function allows us to cope with nontermination (a diverging `while`) and, for languages richer than WHILE, nondeterminism. It also arguably provides a nicer syntax, closer to the judgment rules that are traditionally used in the scientific literature. If we were instead to attempt a recursive definition such as

```
def eval : Stmt → State → State
  | Stmt.skip,              s => s
  | Stmt.assign x a,        s => s[x ↦ a s]
  | Stmt.ifThenElse b S T, s =>
    if b s then eval S s else eval T s
  | S; T,                   s => eval T (eval S s)
  | Stmt.whileDo b S,       s =>
    if b s then eval (Stmt.whileDo b S) (eval S s) else s
```

we would face nontermination of the `Stmt.whileDo` case. Indeed, since the program `Stmt.whileDo (fun _ ↦ True) Stmt.skip` loops forever, trying to evaluate it using `eval` would never return.

Equipped with a big-step semantics, we can reason about concrete programs such as the one defined in Section 9.2 and prove theorems such as the following:

```
theorem sillyLoop_from_1_BigStep :
  (sillyLoop, (fun _ ↦ 0)["x" ↦ 1]) ⟹ (fun _ ↦ 0) :=
  by
    rw [sillyLoop]
    apply BigStep.while_true
    { simp }
    { apply BigStep.seq
      { apply BigStep.skip }
      { apply BigStep.assign } }
    { simp
      apply BigStep.while_false
      simp }
```

9.4 Properties of the Big-Step Semantics

Equipped with a big-step semantics, we can reason about concrete programs, proving theorems relating final states with initial states. Perhaps more interest-

ingly, we can prove properties of the programming language, such as determinism and nontermination.

We start with determinism. It may seem like a trivial property, but it is easy to mistype a rule and obtain nondeterminism. For example, in the rule for assignment, if we mistakenly write `BigStep (Stmt.assign x a, s) (s[y ↦ a s])` with `y` instead of `x`, we suddenly can use the rule to modify any variable `y` we please. In other words, an execution of the program could modify any variable at random. So let us verify that our WHILE language really is deterministic:

```
theorem BigStep_deterministic {Ss l r} (hl : Ss ⟹ l)
    (hr : Ss ⟹ r) :
  l = r
```

The Lean proof is in the demonstration file associated with this chapter. For technical reasons, the pair `(S, s)` is represented by a single variable `Ss`. We content ourselves with an informal proof sketch:

> The proof is by rule induction over `(S, s) ⟹ l`.
>
> Case Skip: To have `(skip, s) ⟹ l` or `(skip, s) ⟹ r`, we need `l = r = s`.
>
> Case Assign: Similar to Skip.
>
> Case Seq: We have the hypotheses `(S, s) ⟹ t`, `(T, t) ⟹ l`, `(S, s) ⟹ t'`, and `(T, t') ⟹ r` and the induction hypotheses `∀r, (S, s) ⟹ r → t = r` and `∀r, (T, t) ⟹ r → l = r`. From the first induction hypothesis together with `(S, s) ⟹ t'`, we derive `t = t'`. From the second induction hypothesis together with `(T, t') ⟹ r`, we derive `l = r`.
>
> Case If-True: Since `b s` is true, `(if b then S else T, s) ⟹ r` can only have been derived using If-True and thus `(S, s) ⟹ r`. The induction hypothesis is `∀r, (S, s) ⟹ r → l = r`. We can apply `(S, s) ⟹ r` to it to obtain `l = r`.
>
> Case If-False: Similar to If-True.
>
> Case While-True: Similar to Seq.
>
> Case While-False: Similar to Skip. □

Given that the WHILE language is deterministic, for the big-step semantics, termination would amount to the following:

```
theorem BigStep_terminates {S s} :
  ∃t, (S, s) ⟹ t
```

This property means that for every statement `S` and state `s`, there exists a state `t` such that executing `S` starting in `s` may terminate in `t`. Because WHILE is deterministic, "may terminate" means the same as "must terminate." However, the property does not hold.

When reasoning about an inductive predicate, it is often convenient to use inversion rules (Section 6.5). Accordingly, we prove the following rules:

```
@[simp] theorem BigStep_skip_Iff {s t} :
  (Stmt.skip, s) ⟹ t ↔ t = s
```

```
@[simp] theorem BigStep_assign_Iff {x a s t} :
  (Stmt.assign x a, s) ⟹ t ↔ t = s[x ↦ a s]

@[simp] theorem BigStep_seq_Iff {S T s u} :
  (S; T, s) ⟹ u ↔ (∃t, (S, s) ⟹ t ∧ (T, t) ⟹ u)

@[simp] theorem BigStep_if_Iff {B S T s t} :
  (Stmt.ifThenElse B S T, s) ⟹ t ↔
  (B s ∧ (S, s) ⟹ t) ∨ (¬ B s ∧ (T, s) ⟹ t)

theorem BigStep_while_Iff {B S s u} :
  (Stmt.whileDo B S, s) ⟹ u ↔
  (∃t, B s ∧ (S, s) ⟹ t ∧ (Stmt.whileDo B S, t) ⟹ u)
  ∨ (¬ B s ∧ u = s)

@[simp] theorem BigStep_while_true_Iff {B S s u}
    (hcond : B s) :
  (Stmt.whileDo B S, s) ⟹ u ↔
  (∃t, (S, s) ⟹ t ∧ (Stmt.whileDo B S, t) ⟹ u)

@[simp] theorem BigStep_while_false_Iff {B S s t}
    (hcond : ¬ B s) :
  (Stmt.whileDo B S, s) ⟹ t ↔ t = s
```

We add most of the rules to the simp set. We omit `BigStep_while_Iff` because it makes `simp` loop.

9.5 Small-Step Semantics

A limitation of big-step semantics is that they do not let us reason about intermediate states. From a judgment `(S, s) ⟹ t`, all we see is the initial state `s` and the final state `t`. This is too coarse-grained to reason about multithreaded programs, where several processes can interact with each other's intermediate states. Moreover, for nondeterministic languages, big-step semantics offer no general way to express termination: A judgment indicates a possibility (executing `S` in state `s` *may* result in state `t`), not a necessity.

Small-step operational semantics provide a finer view. The transition predicate ⇒ has type `Stmt × State → Stmt × State → Prop`. Intuitively, `(S, s) ⇒ (T, t)` means that executing one step of program `S` in state `s` leaves the program `T` to be executed, in state `t`. If there is nothing left to be executed, we put `skip`.

An *execution* is a finite or infinite chain $(S_0, s_0) \Rightarrow (S_1, s_1) \Rightarrow \cdots$ of "small" ⇒ steps. A pair `(S, s)` is called a *configuration*; it is *final* if no transition of the form `(S, s) ⇒ (T, t)` is possible, for any `(T, t)`. A possible execution follows:

```
   (x := x + y; y := 0, [x ↦ 3, y ↦ 5])
⇒ (skip; y := 0,       [x ↦ 8, y ↦ 5])
⇒ (y := 0,             [x ↦ 8, y ↦ 5])
⇒ (skip,               [x ↦ 8, y ↦ 0])
```

If we take the analogy of a computer processor, the `S` component of a configuration `(S, s)` can be thought of as a program counter, which indicates which

instructions should be executed next. The step-by-step execution of a program resembles running a program in a debugger with a breakpoint at each step.

The valid small-step judgments are given by derivation rules:

$$\frac{}{(\texttt{x := a, s}) \Rightarrow (\texttt{skip, s}[\texttt{x} \mapsto \texttt{a s}])}\ \textsc{Assign}$$

$$\frac{(\texttt{S, s}) \Rightarrow (\texttt{S', s'})}{(\texttt{S; T, s}) \Rightarrow (\texttt{S'; T, s'})}\ \textsc{Seq-Step}$$

$$\frac{}{(\texttt{skip; T, s}) \Rightarrow (\texttt{T, s})}\ \textsc{Seq-Skip}$$

$$\frac{}{(\texttt{if b then S else T, s}) \Rightarrow (\texttt{S, s})}\ \textsc{If-True} \quad \text{if } \texttt{b s} \text{ is true}$$

$$\frac{}{(\texttt{if b then S else T, s}) \Rightarrow (\texttt{T, s})}\ \textsc{If-False} \quad \text{if } \texttt{b s} \text{ is false}$$

$$\frac{}{(\texttt{while b do S, s}) \Rightarrow (\texttt{if b then (S; while b do S) else skip, s})}\ \textsc{While}$$

The rules are reminiscent of the tennis game transition system of Section 6.1.2. These, too, specified small steps: from `0–0` to `15–0`, to `15–15`, and so on.

Unlike in the big-step semantics, there is no rule for `skip` in the small-step semantics. This is because a configuration of the form `(skip, s)` is considered final; `skip` is understood as the statement whose execution is trivial. By inspection of the rules, we can convince ourselves that a configuration is final if and only if its first component is `skip`.

Two rules concern sequential composition `S; T`. The first rule is applicable if some progress can be made executing `S`. But if `S` is `skip`, no progress can be made and the second rule applies.

The rules for `if` check the condition `b` and, depending on its truth value, put the `then` or the `else` branch as the remaining computation to perform.

For the `while` loop, there is a single unconditional rule that expands one iteration of the loop, introducing an `if` statement. It is then the role of the If-True and If-False rules to process the `if`. In the If-True case, we eventually reach the `while` loop again. This can continue forever for infinite loops.

In Lean, the small-step semantics is specified as follows:

```
inductive SmallStep : Stmt × State → Stmt × State → Prop
    where
  | assign (x a s) :
    SmallStep (Stmt.assign x a, s) (Stmt.skip, s[x ↦ a s])
  | seq_step (S S' T s s') (hS : SmallStep (S, s) (S', s')) :
    SmallStep (S; T, s) (S'; T, s')
  | seq_skip (T s) :
    SmallStep (Stmt.skip; T, s) (T, s)
  | if_true (B S T s) (hcond : B s) :
    SmallStep (Stmt.ifThenElse B S T, s) (S, s)
  | if_false (B S T s) (hcond : ¬ B s) :
    SmallStep (Stmt.ifThenElse B S T, s) (T, s)
  | whileDo (B S s) :
```

```
    SmallStep (Stmt.whileDo B S, s)
      (Stmt.ifThenElse B (S; Stmt.whileDo B S) Stmt.skip, s)
```

Based on a small-step semantics, we can *define* a big-step semantics as follows:

$$(S, s) \Longrightarrow t \quad \text{if and only if} \quad (S, s) \Rightarrow^* (\texttt{skip}, t)$$

where p^* denotes the reflexive transitive closure (RTC) of a binary predicate p. Alternatively, if we have already defined a big-step semantics, we can *prove* the above equivalence theorem to *validate* our definitions.

The main disadvantage of small-step semantics is that we now have two predicates, $\Rightarrow$ and $\Rightarrow^*$, and the derivation rules and proofs tend to be more complicated than with big steps. This is clearly visible on the following example, in which we need to apply the theorem

$$\texttt{RTC.head} : \texttt{?R ?a ?b} \to \texttt{RTC ?R ?b ?c} \to \texttt{RTC ?R ?a ?c}$$

once for each small step:

```
theorem sillyLoop_from_1_SmallStep :
  (sillyLoop, (fun _ ↦ 0)["x" ↦ 1]) ⇒*
  (Stmt.skip, (fun _ ↦ 0)) :=
  by
    rw [sillyLoop]
    apply RTC.head
    { apply SmallStep.whileDo }
    { apply RTC.head
      { apply SmallStep.if_true
        simp }
      { apply RTC.head
        { apply SmallStep.seq_step
          apply SmallStep.seq_skip }
        { apply RTC.head
          { apply SmallStep.seq_step
            apply SmallStep.assign }
          { apply RTC.head
            { apply SmallStep.seq_skip }
            { apply RTC.head
              { apply SmallStep.whileDo }
              { apply RTC.head
                { apply SmallStep.if_false
                  simp }
                { simp
                  apply RTC.refl } } } } } } }
```

9.6 Properties of the Small-Step Semantics

We can prove that a configuration `(S, s)` is final if and only if `S = skip`. Doing so ensures that we have not forgotten a derivation rule and hence that the small-steps semantics cannot get stuck. The theorem statement is as follows:

```
theorem SmallStep_final (S s) :
  (¬ ∃T t, (S, s) ⇒ (T, t)) ↔ S = Stmt.skip
```

The proof is by structural induction on `S`.

Like the big-step semantics, the small-step semantics is deterministic:

```
theorem SmallStep_deterministic {Ss Ll Rr}
    (hl : Ss ⇒ Ll) (hr : Ss ⇒ Rr) :
  Ll = Rr
```

The proof is by rule induction on `hl` or `hr`.

For the small-step semantics, a configuration (S_0, s_0) terminates if all executions starting in it are finite: $(S_0, s_0) \Rightarrow (S_1, s_1) \Rightarrow \cdots \Rightarrow (S_n, s_n)$. It is nonterminating if there exists an infinite chain $(S_0, s_0) \Rightarrow (S_1, s_1) \Rightarrow \cdots$. The programming language as a whole is terminating if and only if all its configurations terminate. It is easy to show that the WHILE language is nonterminating, by taking S_0 `:= Stmt.whileDo (fun _ ↦ True) Stmt.skip`. For any s_0, we then have

$$(S_0, s_0) \Rightarrow (S_0, s_0) \Rightarrow (S_0, s_0) \Rightarrow \cdots$$

We can define inversion rules about the small-step semantics, such as these:

```
theorem SmallStep_skip {S s t} :
  ¬ ((Stmt.skip, s) ⇒ (S, t))

@[simp] theorem SmallStep_seq_Iff {S T s Ut} :
  (S; T, s) ⇒ Ut ↔
  (∃S' t, (S, s) ⇒ (S', t) ∧ Ut = (S'; T, t))
  ∨ (S = Stmt.skip ∧ Ut = (T, s))

@[simp] theorem SmallStep_if_Iff {B S T s Us} :
  (Stmt.ifThenElse B S T, s) ⇒ Us ↔
  (B s ∧ Us = (S, s)) ∨ (¬ B s ∧ Us = (T, s))
```

A more fundamental result is the equivalence between the big-step and the small-step semantics:

```
theorem BigStep_Iff_RTC_SmallStep {Ss t} :
  Ss ⟹ t ↔ Ss ⇒* (Stmt.skip, t)
```

Recall that `⇒*` denotes the reflexive transitive closure of the small-step predicate `⇒`. The theorem's proof is beyond the scope of this course. We refer to Chapter 7 of *Concrete Semantics: With Isabelle/HOL* [23] or to the demonstration file for this chapter.

Chapter 10

Hoare Logic

If operational semantics corresponds to an idealized interpreter, *Hoare logic* corresponds to an idealized verifier. Hoare logic can be used to specify the semantics of a programming language, but it is particularly convenient to reason about concrete programs and prove them correct. It is named after its inventor, Charles Antony Richard (Tony) Hoare. Hoare logic is also called *axiomatic semantics*.

This chapter is heavily inspired by Chapter 12 of *Concrete Semantics: With Isabelle/HOL* [23].

10.1 Hoare Triples

Hoare logic is a framework for deducing valid correctness formulas in a mechanical way, using a set of derivation rules. It allows us to reason directly about a program's syntax, without concerning ourselves with its operational semantics. The approach is mechanical in the sense that the applicability of a derivation rule can easily be checked.

We start by introducing Hoare logic abstractly, without any connection to Lean. In a second step, we will see how we can embed Hoare logic judgments in Lean. The basic judgments of Hoare logic are called *Hoare triples*. They have the form `{P} S {Q}`, where `S` is a WHILE statement, and `P` and `Q` are logical formulas over the program variables. For the moment, we imagine the formulas as syntactic objects built using the familiar connectives and quantifiers. The intended meaning of a Hoare triple is as follows:

> If the *precondition* `P` is true before `S` is executed and the execution terminates normally, the *postcondition* `Q` is true at termination.

This is a *partial correctness* statement: The program is correct *if* it terminates normally; otherwise, the program might behave arbitrarily. For WHILE programs, the only way not to terminate normally is to enter an infinite loop. For other programming languages, infinite recursion and run-time errors such as division by zero may also result in divergence or abnormal termination.

Intuitively, all of the Hoare triples below should be valid:

$$\{\mathtt{True}\}\ \mathtt{b} := 4\ \{\mathtt{b} = 4\}$$

$$\{\mathtt{a} = 2\}\ \mathtt{b} := 2 * \mathtt{a}\ \{\mathtt{a} = 2 \wedge \mathtt{b} = 4\}$$

$$\{\mathtt{b} \geq 5\}\ \mathtt{b} := \mathtt{b} + 1\ \{\mathtt{b} \geq 6\}$$

$$\{\texttt{False}\}\ \texttt{skip}\ \{\texttt{b = 10}\}$$

$$\{\texttt{True}\}\ \texttt{while i} \neq \texttt{10 do i := i + 1}\ \{\texttt{i = 10}\}$$

The first three Hoare triples should be fairly natural. The fourth triple is vacuously true, since the precondition `False` can never be met. The part "If the precondition `P` is true" of the definition of Hoare triple is always false; hence the triple is true. The triple is equivalent to the proposition `False` $\rightarrow$ `b = 10`, which holds for any value of `b`. As for the fifth triple, there are no guarantees that control will escape the loop (if `i > 10` initially), but if it does escape, then the loop's condition must be false and hence we have the postcondition `i = 10`.

The following triples are bizarre but interesting:

$$\{\texttt{False}\}\ \texttt{S}\ \{\texttt{True}\}$$

$$\{\texttt{False}\}\ \texttt{S}\ \{\texttt{False}\}$$

$$\{\texttt{True}\}\ \texttt{S}\ \{\texttt{True}\}$$

$$\{\texttt{True}\}\ \texttt{S}\ \{\texttt{False}\}$$

The first two triples are true of any statement `S` (and therefore pointless): The precondition is never satisfied, so any postcondition holds vacuously. The third triple is always true as well, regardless of `S`. The fourth triple is true if `S` never terminates (e.g., `S := while True do skip`); otherwise, it is false.

10.2 Hoare Rules

Below we give a complete set of derivation rules for reasoning about WHILE programs:

$$\frac{}{\{\texttt{P}\}\ \texttt{skip}\ \{\texttt{P}\}}\ \textsc{Skip}$$

$$\frac{}{\{\texttt{Q[a/x]}\}\ \texttt{x := a}\ \{\texttt{Q}\}}\ \textsc{Assign}$$

$$\frac{\{\texttt{P}\}\ \texttt{S}\ \{\texttt{R}\} \qquad \{\texttt{R}\}\ \texttt{T}\ \{\texttt{Q}\}}{\{\texttt{P}\}\ \texttt{S; T}\ \{\texttt{Q}\}}\ \textsc{Seq}$$

$$\frac{\{\texttt{P} \wedge \texttt{B}\}\ \texttt{S}\ \{\texttt{Q}\} \qquad \{\texttt{P} \wedge \neg\texttt{B}\}\ \texttt{T}\ \{\texttt{Q}\}}{\{\texttt{P}\}\ \texttt{if B then S else T}\ \{\texttt{Q}\}}\ \textsc{If}$$

$$\frac{\{\texttt{P} \wedge \texttt{B}\}\ \texttt{S}\ \{\texttt{P}\}}{\{\texttt{P}\}\ \texttt{while B do S}\ \{\texttt{P} \wedge \neg\texttt{B}\}}\ \textsc{While}$$

$$\frac{\texttt{P}' \rightarrow \texttt{P} \qquad \{\texttt{P}\}\ \texttt{S}\ \{\texttt{Q}\} \qquad \texttt{Q} \rightarrow \texttt{Q}'}{\{\texttt{P}'\}\ \texttt{S}\ \{\texttt{Q}'\}}\ \textsc{Conseq}$$

In the ASSIGN rule, the expression `Q[a/x]` denotes the condition `Q` in which all occurrences of `x` are replaced by `a`. The rule is sometimes presented as

$$\frac{}{\{\texttt{Q[a]}\}\ \texttt{x := a}\ \{\texttt{Q[x]}\}}\ \textsc{Assign}$$

where `[x]` factors out the occurrences of `x` in `Q`.

The ASSIGN rule may seem counterintuitive because it works backwards: From the postcondition, it computes a precondition. Nevertheless, it correctly captures the semantics of the assignment statement, as illustrated below:

$$\{0 = 0\}\ x := 0\ \{x = 0\}$$
$$\{0 = 0 \wedge y = 5\}\ x := 0\ \{x = 0 \wedge y = 5\}$$
$$\{x + 1 \geq 5\}\ x := x + 1\ \{x \geq 5\}$$

Using elementary arithmetic, we can simplify the computed preconditions; for example, $0 = 0$ is equivalent to `True`, and $x + 1 \geq 5$ is equivalent to $x \geq 4$.

The SEQ rule requires us to come up with an intermediate condition `R` that holds after executing `S` and before executing `T`. Here is an example of SEQ in action:

$$\dfrac{\dfrac{}{\{a = 2\}\ b := a\ \{b = 2\}}\ \textsc{Assign} \qquad \dfrac{}{\{b = 2\}\ c := b\ \{c = 2\}}\ \textsc{Assign}}{\{a = 2\}\ b := a;\ c := b\ \{c = 2\}}\ \textsc{Seq}$$

The WHILE rule is the most intricate. The condition `P` is called an *invariant*. It is the pre- and the postcondition of the loop itself but also of its body. The body's precondition is strengthened with the knowledge that `B` must be true before executing the body. Similarly, the loop's postcondition is strengthened with the knowledge that `B` must be false when the loop exits.

Consider an execution of a loop with n iterations. Suppose that the initial state is s_0 and that the state after the ith iteration of the loop is s_i. Then the following conditions will hold:

$$P\ s_0 \qquad P\ s_1 \wedge B\ s_1 \qquad \cdots \qquad P\ s_{n-1} \wedge B\ s_{n-1} \qquad P\ s_n \wedge \neg\ B\ s_n$$

If $n = 0$, we immediately have that $P\ s_0 \wedge \neg\ B\ s_0$ and never enter the loop.

CONSEQ is the only rule that has logical formulas among its premises, as opposed to Hoare triples. These conditions must be discharged, whether using pen and paper or a proof assistant. CONSEQ can be used to strengthen a precondition (i.e., make it more restrictive), weaken a postcondition (i.e., make it less restrictive), or both. An example derivation follows:

$$\dfrac{x > 8 \rightarrow x > 4 \qquad \dfrac{}{\{x > 4\}\ y := x\ \{y > 4\}}\ \textsc{Assign} \qquad y > 4 \rightarrow y > 0}{\{x > 8\}\ y := x\ \{y > 0\}}\ \textsc{Conseq}$$

Reading the tree from top to bottom, we have strengthened the triple's precondition from $x > 4$ to $x > 8$ and weakened the postcondition from $y > 4$ to $y > 0$. We can also read the tree from the bottom up: To prove the triple $\{x > 8\}\ y := x\ \{y > 0\}$, it suffices to prove $\{x > 4\}\ y := x\ \{y > 4\}$, where the precondition is weakened and the postcondition is strengthened.

Except for CONSEQ, the rules are *syntax-driven*: We know which rule to apply in each case, simply by inspecting the statement at hand. For an assignment, we apply ASSIGN; for a `while` loop, we apply WHILE; and so on.

The rules SEQ, IF, and CONSEQ are *bidirectional*: Their conclusions are of the form $\{P\}\ \ldots\ \{Q\}$ for distinct mathematical variables `P`, `Q`. This can make them convenient to apply. By combining the other rules with CONSEQ, we can derive bidirectional variants:

$$\frac{P \to Q}{\{P\}\ \texttt{skip}\ \{Q\}}\ \textsc{Skip}'$$

$$\frac{P \to Q[a/x]}{\{P\}\ x := a\ \{Q\}}\ \textsc{Assign}'$$

$$\frac{\{P \wedge B\}\ S\ \{P\} \qquad P \wedge \neg B \to Q}{\{P\}\ \texttt{while}\ B\ \texttt{do}\ S\ \{Q\}}\ \textsc{While}'$$

As an exercise, you could try to derive each of these rules from Skip, Assign, or While in conjunction with Conseq.

10.3 A Semantic Approach to Hoare Logic

A natural way to encode Hoare logic in Lean would be to proceed as we have done for the big- and small-step semantics: Define a syntactic notion of Hoare triple and an inductive predicate, with one introduction rule for each core Hoare rule; to represent pre- and postconditions, use predicates on states (`State → Prop`). Then we could prove *soundness* with respect to the big-step semantics, meaning the following: Whenever `{P} S {Q}` is derivable, if `P s` and `(S, s) ⟹ t`, then `Q t`. This is merely a logical rendition of the intuitive meaning of a Hoare triple:

> If the precondition `P` is true before `S` is executed (i.e., `P s`) and the execution terminates normally (i.e., `(S, s) ⟹ t`), the postcondition `Q` is true at termination (i.e., `Q t`).

Instead of pursuing this approach, we propose to define Hoare triples semantically in Lean, in terms of the big-step semantics, so that they are *correct by definition*. Then we will derive the Hoare rules as theorems, instead of stating them as introduction rules. In conjunction with the use of predicates to represent formulas, this approach is resolutely semantic.

Hoare triples (for partial correctness) are defined as follows:

```
def PartialHoare (P : State → Prop) (S : Stmt)
    (Q : State → Prop) : Prop :=
  ∀s t, P s → (S, s) ⟹ t → Q t
```

Instead of writing `PartialHoare P S Q`, we introduce some syntactic sugar to allow `{* P *} (S) {* Q *}`, which is closer to the informal syntax `{P} S {Q}`.

The core Hoare rules are stated as follows:

```
theorem skip_intro {P} :
  {* P *} (Stmt.skip) {* P *}

theorem assign_intro (P) {x a} :
  {* fun s ↦ P (s[x ↦ a s]) *} (Stmt.assign x a) {* P *}

theorem seq_intro {P Q R S T} (hS : {* P *} (S) {* Q *})
    (hT : {* Q *} (T) {* R *}) :
  {* P *} (S; T) {* R *}
```

```
theorem if_intro {B P Q S T}
    (hS : {* fun s ↦ P s ∧ B s *} (S) {* Q *})
    (hT : {* fun s ↦ P s ∧ ¬ B s *} (T) {* Q *}) :
  {* P *} (Stmt.ifThenElse B S T) {* Q *}

theorem while_intro (P) {B S}
    (h : {* fun s ↦ P s ∧ B s *} (S) {* P *}) :
  {* P *} (Stmt.whileDo B S) {* fun s ↦ P s ∧ ¬ B s *}

theorem consequence {P P' Q Q' S}
    (h : {* P *} (S) {* Q *}) (hp : ∀s, P' s → P s)
    (hq : ∀s, Q s → Q' s) :
  {* P' *} (S) {* Q' *}
```

All of the above theorems have proofs based on the big-step semantics. Some of the triples—for example, the precondition in `assign_intro`—need to refer to the state. We then use an anonymous function to access it. Recall that `P` and `fun s ↦ P s` are equal (by η-conversion). Moreover, for a premise written informally as $P \to Q$, in Lean we must write `∀s, P s → Q s`. As in Chapter 9, the syntax `s[x ↦ n]` in the assignment rule denotes the state that is identical to `s` except that it maps `x` to `n`.

The following convenience rules can be derived from the core rules:

```
theorem consequence_left (P') {P Q S}
    (h : {* P *} (S) {* Q *}) (hp : ∀s, P' s → P s) :
  {* P' *} (S) {* Q *}

theorem consequence_right (Q) {Q' P S}
    (h : {* P *} (S) {* Q *}) (hq : ∀s, Q s → Q' s) :
  {* P *} (S) {* Q' *}

theorem skip_intro' {P Q} (h : ∀s, P s → Q s) :
  {* P *} (Stmt.skip) {* Q *}

theorem assign_intro' {P Q x a}
    (h : ∀s, P s → Q (s[x ↦ a s])):
  {* P *} (Stmt.assign x a) {* Q *}

theorem seq_intro' {P Q R S T} (hT : {* Q *} (T) {* R *})
    (hS : {* P *} (S) {* Q *}) :
  {* P *} (S; T) {* R *}

theorem while_intro' {B P Q S} (I)
    (hS : {* fun s ↦ I s ∧ B s *} (S) {* I *})
    (hP : ∀s, P s → I s)
    (hQ : ∀s, ¬ B s → I s → Q s) :
  {* P *} (Stmt.whileDo B S) {* Q *}
```

Using the bidirectional `assign_intro'`, we can derive a forward version of the assignment rule:

```
theorem assign_intro_forward (P) {x a} :
  {* P *}
  (Stmt.assign x a)
  {* fun s ↦ ∃n₀, P (s[x ↦ n₀]) ∧ s x = a (s[x ↦ n₀]) *} :=
  by
    apply assign_intro'
    intro s hP
    apply Exists.intro (s x)
    simp [*]
```

The variable `n₀` stands for the value of `x` before the assignment. Hence, in the postcondition, `s[x ↦ n₀]` denotes the state before the assignment. Since `P` is a precondition, we have `P (s[x ↦ n₀])`. In addition, the new value of `x`, given by `s x`, must be equal to the value of the expression `a` evaluated in the old state `s[x ↦ n₀]`.

The forward rule is less convenient than the backward rule, because the postcondition contains an existential quantifier. It is possible to state a backward rule in a similar style, revealing a hidden symmetry:

```
theorem assign_intro_backward (Q) {x a} :
  {* fun s ↦ ∃n', Q (s[x ↦ n']) ∧ n' = a s *}
  (Stmt.assign x a)
  {* Q *}
```

Notice that this existential quantifier can be eliminated using a one-point rule (Section 4.3). We then get the familiar backward rule `assign_intro`, with `fun s ↦ P (s[x ↦ a s])` as the precondition.

10.4 First Program: Exchanging Two Variables

Let us employ Hoare logic to verify our first program: a three-line program that exchanges the values of its variables `a` and `b`, using `t` for temporary storage. The program is defined as follows:

```
def SWAP : Stmt :=
  Stmt.assign "t" (fun s ↦ s "a");
  Stmt.assign "a" (fun s ↦ s "b");
  Stmt.assign "b" (fun s ↦ s "t")
```

The correctness statement is as follows:

```
theorem SWAP_correct (a₀ b₀ : ℕ) :
  {* fun s ↦ s "a" = a₀ ∧ s "b" = b₀ *}
  (SWAP)
  {* fun s ↦ s "a" = b₀ ∧ s "b" = a₀ *}
```

The logical variables `a₀` and `b₀` "freeze" the initial value of the program variables `a` and `b` so that we can refer to them in the postcondition. After all, it would make no sense to use `fun s ↦ s "a" = s "b" ∧ s "b" = s "a"` as the postcondition.

The correctness proof follows:

```
  by
    apply PartialHoare.seq_intro'
```

```
  apply PartialHoare.seq_intro'
  apply PartialHoare.assign_intro
  apply PartialHoare.assign_intro
  apply PartialHoare.assign_intro'
  aesop
```

The applications of the sequential composition and assignment rules are guided by the program's syntax. There are two sequential compositions and three assignments in the program and therefore as many invocations of the corresponding rules. We end up with a very aesthetically challenged subgoal:

```
⊢ ∀s : State,
    s "a" = a₀ ∧ s "b" = b₀ →
    s["t" ↦ s "a"]["a" ↦ s["t" ↦ s "a"] "b"]
      ["b" ↦ s["t" ↦ s "a"]["a" ↦ s["t" ↦ s "a"] "b"] "t"] "a" = b₀ ∧
    s["t" ↦ s "a"]["a" ↦ s["t" ↦ s "a"] "b"]
      ["b" ↦ s["t" ↦ s "a"]["a" ↦ s["t" ↦ s "a"] "b"] "t"] "b" = a₀
```

Fortunately, `simp [*] at *` can reduce the subgoal dramatically, and `aesop` can even prove it fully automatically.

10.5 Second Program: Adding Two Numbers

Our second example computes `m + n`, leaving the result in `m`, using only these primitive operations: `k + 1`, `k - 1`, and `k ≠ 0` (for arbitrary `k`):

```
def ADD : Stmt :=
  Stmt.whileDo (fun s ↦ s "n" ≠ 0)
    (Stmt.assign "n" (fun s ↦ s "n" - 1);
     Stmt.assign "m" (fun s ↦ s "m" + 1))
```

Because of the presence of a `while` loop, the proof is more involved:

```
theorem ADD_correct (n₀ m₀ : ℕ) :
  {* fun s ↦ s "n" = n₀ ∧ s "m" = m₀ *}
  (ADD)
  {* fun s ↦ s "n" = 0 ∧ s "m" = n₀ + m₀ *} :=
  PartialHoare.while_intro' (fun s ↦ s "n" + s "m" = n₀ + m₀)
    (by
      apply PartialHoare.seq_intro'
      { apply PartialHoare.assign_intro }
      { apply PartialHoare.assign_intro'
        aesop })
    (by aesop)
    (by aesop)
```

The first step is to apply the derived `while` rule with a loop invariant. Our invariant is that the sum of the program variables `n` and `m` must be equal to the desired mathematical result `n₀ + m₀`, where `n₀` and `m₀` correspond to the initial values of `n` and `m`, as required by the precondition.

How did we come up with this invariant? Even for a simple loop, finding a suitable invariant can be challenging. The difficulty is that the invariant must

1. be true before we enter the loop;
2. remain true after each iteration of the loop if it was true before the iteration;
3. be strong enough to imply the desired loop postcondition.

An invariant such as `True` meets requirements 1 and 2 but usually not 3. Similarly, `False` meets requirements 2 and 3 but not 1. In practice, invariants tend to be of the form

work done + *work remaining* = *desired result*

where + stands for some appropriate operator (not necessarily addition). When we enter the loop, *work done* will often be 0 (or some other appropriate "zero" value), and the invariant becomes

work remaining = *desired result*

This invariant would have to be provable at the beginning of the loop—either from the postcondition of the previous statement or from the desired precondition of the entire program if there is no previous statement. When we exit the loop, *work remaining* should be 0 (or some variant thereof), and the invariant becomes

work done = *desired result*

Often, *work done* takes the form of a variable in which we accumulate the result, whereas *work remaining* is similar to *desired result* but depends on program variables whose values change inside the loop and accounts for *work done*.

For the `ADD` program's loop, the *work done* is `m`, the *work remaining* is `n`, and the *desired result* is $n_0 + m_0$. When entering the loop, the invariant $m + n = n_0 + m_0$ holds because then $m = m_0$ and $n = n_0$. (Unusually, the *work done* is not 0 for this example, because we reuse the input `m` as our accumulator, as an optimization.) When exiting the loop, we have that $n = 0$, so the invariant becomes $m = n_0 + m_0$. We can retrieve the result from `m`.

The `while_intro'` theorem is used directly as a proof term. It gives rise to three subgoals. The proofs that the invariant is implied by the desired precondition and that it implies the desired postcondition are trivial: They consist of a call to `aesop`. The only nontrivial subgoal is the condition that executing the body maintains the invariant.

For this example, Hoare logic really helps. Reasoning directly about the operational semantics would be inconvenient, because we would need induction to reason about the `while` loop. With Hoare logic, this induction is performed out once and for all in the proof of the `while_intro` rule.

10.6 A Verification Condition Generator

Verification condition generators (VCGs) are programs that apply Hoare logic rules, producing *verification conditions* that must be proved manually. We can think of them as mechanical civil servants that take care of the Hoare logic bureaucracy. As users, we must provide strong enough loop invariants as annotations in our programs. Hundreds of program verification tools are based on these principles.

VCGs typically work backwards from the postcondition, using *backward* rules—rules stated to have an arbitrary `Q` as their postcondition. This works well because the central rule of Hoare logic—the assignment rule—is backward.

We can use Lean's metaprogramming framework to define a simple VCG. First, we introduce a constant called `Stmt.invWhileDo` that carries a user-supplied invariant `I` in addition to the loop condition `B` and the body `S`:

```
def Stmt.invWhileDo (I B : State → Prop) (S : Stmt) : Stmt :=
  Stmt.whileDo B S
```

We provide two Hoare rules for the construct: a backward rule and a bidirectional rule. Both are justified in terms of the bidirectional `while_intro'` rule:

```
theorem invWhile_intro {B I Q S}
    (hS : {* fun s ↦ I s ∧ B s *} (S) {* I *})
    (hQ : ∀s, ¬ B s → I s → Q s) :
  {* I *} (Stmt.invWhileDo I B S) {* Q *} :=
  while_intro' I hS (by aesop) hQ

theorem invWhile_intro' {B I P Q S}
    (hS : {* fun s ↦ I s ∧ B s *} (S) {* I *})
    (hP : ∀s, P s → I s) (hQ : ∀s, ¬ B s → I s → Q s) :
  {* P *} (Stmt.invWhileDo I B S) {* Q *} :=
  while_intro' I hS hP hQ
```

The above rules simply use the invariant annotation as their invariant. When using the framework, we will have to be careful to annotate all `while` loops with suitable invariants. If we specify a wrong invariant, we will face unprovable subgoals, indicating that we must revise the invariant.

The code of the VCG is fairly concise:

```
def matchPartialHoare : Expr → Option (Expr × Expr × Expr)
  | (Expr.app (Expr.app (Expr.app
      (Expr.const ''PartialHoare _) P) S) Q) =>
    Option.some (P, S, Q)
  | _ =>
    Option.none

partial def vcg : TacticM Unit :=
  do
    let goals ← getUnsolvedGoals
    if goals.length != 0 then
      let target ← getMainTarget
      match matchPartialHoare target with
      | Option.none           => return
      | Option.some (P, S, Q) =>
        if Expr.isAppOfArity S ''Stmt.skip 0 then
          if Expr.isMVar P then
            applyConstant ''PartialHoare.skip_intro
          else
            applyConstant ''PartialHoare.skip_intro'
        else if Expr.isAppOfArity S ''Stmt.assign 2 then
          if Expr.isMVar P then
            applyConstant ''PartialHoare.assign_intro
          else
```

```
          applyConstant ``PartialHoare.assign_intro'
        else if Expr.isAppOfArity S ``Stmt.seq 2 then
          andThenOnSubgoals
            (applyConstant ``PartialHoare.seq_intro') vcg
        else if Expr.isAppOfArity S ``Stmt.ifThenElse 3 then
          andThenOnSubgoals
            (applyConstant ``PartialHoare.if_intro) vcg
        else if Expr.isAppOfArity S ``Stmt.invWhileDo 3 then
          if Expr.isMVar P then
            andThenOnSubgoals
              (applyConstant ``PartialHoare.invWhile_intro) vcg
          else
            andThenOnSubgoals
              (applyConstant ``PartialHoare.invWhile_intro')
              vcg
        else
          failure

elab "vcg" : tactic =>
  vcg
```

The VCG extracts the first goal's target and inspects it. If it is a Hoare triple, the VCG inspects its precondition `P` and statement `S`. If the precondition is a metavariable (e.g., `?P`), the VCG applies a backward rule (via the `applyConstant` function we defined in Section 8.7) if there exists one because this will instantiate the metavariable. Otherwise, a bidirectional rule is used, with an arbitrary variable as its precondition, which can be matched against the target's precondition. For `while` loops, we only consider programs that use `Stmt.invWhileDo`, because we cannot guess the invariant programmatically.

The VCG calls itself recursively on all newly emerging subgoals for the compound statements (via the `andThenOnSubgoals` function we defined in Section 8.7).

10.7 Second Program Revisited: Adding Two Numbers

Using the verification condition generator, we can revisit the correctness proof for the `ADD` program presented above:

```
theorem ADD_correct_vcg (n₀ m₀ : ℕ) :
  {* fun s ↦ s "n" = n₀ ∧ s "m" = m₀ *}
  (ADD)
  {* fun s ↦ s "n" = 0 ∧ s "m" = n₀ + m₀ *} :=
show {* fun s ↦ s "n" = n₀ ∧ s "m" = m₀ *}
     (Stmt.invWhileDo (fun s ↦ s "n" + s "m" = n₀ + m₀)
        (fun s ↦ s "n" ≠ 0)
        (Stmt.assign "n" (fun s ↦ s "n" - 1);
         Stmt.assign "m" (fun s ↦ s "m" + 1)))
     {* fun s ↦ s "n" = 0 ∧ s "m" = n₀ + m₀ *} from
  by
    vcg <;>
     aesop
```

First, we use `show` to annotate the `while` loop with an invariant. Recall that the `show` command restates the goal in a computationally equivalent way. Here, we use this facility to replace `Stmt.whileDo` by `Stmt.invWhileDo`, which equals `Stmt.whileDo` by definition. The program and its pre- and postconditions are otherwise the same as in the theorem statement.

We invoke `vcg` as the first proof step. This will apply all the necessary Hoare rules and leave us with some subgoals, which are no match for `aesop`.

10.8 Hoare Triples for Total Correctness

The focus so far in this chapter has been on partial correctness. When we state the Hoare triple `{P} S {Q}`, we claim that the final state will satisfy `Q` *if* the program `S` terminates, but we say nothing when `S` does not terminate. In particular, we can prove any postcondition for the diverging program `while True do skip`. This is admittedly too liberal: If you are asked to program a sorting algorithm at an exam, you should certainly not give `while True do skip` as your answer.

Total correctness is a stronger notion that asserts, in addition to partial correctness, that the program terminates normally. We first focused on partial correctness because it is simpler and because it is a necessary component of total correctness anyway.

The Hoare triples for total correctness have the form `[P] S [Q]`, with the following intended meaning:

> If the precondition `P` holds before `S` is executed, the execution terminates normally and the postcondition `Q` holds in the final state.

For deterministic programs, this can be expressed equivalently as follows:

> If the precondition `P` holds before `S` is executed, there exists a state in which execution terminates normally and the postcondition `Q` holds in that state.

Here is an example Hoare triple that is valid:

$$\texttt{[i} \leq \texttt{10] while i < 10 do i := i + 1 [i = 10]}$$

For the WHILE language, the distinction between partial and total correctness only concerns `while` loops (and programs containing them). The Hoare rule for `while` must now be annotated by a *variant* `v`—a natural number that decreases by one or more with each iteration:

$$\frac{\texttt{[P} \wedge \texttt{b} \wedge \texttt{v = v}_0\texttt{] S [P} \wedge \texttt{v < v}_0\texttt{]}}{\texttt{[P] while b do S [P} \wedge \neg\texttt{b]}}\ \textsc{While-Var}$$

Here, $\texttt{v}_0$ is a logical variable that freezes `v`'s initial value and whose scope is the entire premise, whereas `v` is a mathematical variable (like `P`, `b`, and `S`). For the example above, we could take `10 - i` as the variant (or `50 - i` or `1024 - i * i`).

Consider an execution $\texttt{s}_0, \texttt{s}_1, \ldots, \texttt{s}_{n-1}, \texttt{s}_n$ corresponding to n loop iterations, as in Section 10.2. The following conditions will hold:

$$\begin{array}{lllllll} \texttt{P s}_0 & \texttt{P s}_1 \wedge \texttt{b s}_1 & \cdots & \texttt{P s}_{n-1} \wedge \texttt{b s}_{n-1} & \texttt{P s}_n \wedge \neg\,\texttt{b s}_n \\ \texttt{v s}_0 & > \texttt{v s}_1 & > \cdots > & \texttt{v s}_{n-1} & > \texttt{v s}_n \end{array}$$

Chapter 11

Denotational Semantics

We now review a third way to specify the semantics of a programming language: denotational semantics. A denotational semantics directly specifies the meaning of programs as a mathematical object. If an operational semantics corresponds to an idealized interpreter and a Hoare logic corresponds to an idealized verifier, then a denotational semantics corresponds to an idealized compiler: a compiler that translates the source program not to assembly language but directly to mathematics.

The core of this chapter is modeled closely after Chapter 11 of *Concrete Semantics: With Isabelle/HOL* [23].

11.1 Compositionality

A *denotational semantics* defines the meaning of each program as a mathematical object. Abstractly, it can be viewed as a function

$$[\![\ ]\!] : \text{syntax} \to \text{semantics}$$

A key property of denotational semantics is *compositionality*: The meaning of a compound statement should be defined in terms of the meaning of its components. Consider the straightforward definition

```
⟦S⟧ = {(s, t) | (S, s) ⟹ t}
```

in terms of the big-step semantics ($\Longrightarrow$). Actually, because Lean supports only variables on the left-hand side of the vertical bar, we must write

```
⟦S⟧ = {st | (S, Prod.fst st) ⟹ Prod.snd st}
```

This definition specifies the desired semantics, but it does not qualify as a denotational semantics due to lack of compositionality: The meaning of the compound statements (sequential composition, `if–then–else`, and `while`) is given directly, without using the denotation of their components.

A fully compositional definition allows us to reason equationally about programs, which is often more convenient than using the introduction, elimination, and inversion principles of $\Longrightarrow$. In essence, we want structurally recursive equations of the form

$$\begin{aligned}
[\![\texttt{S ; T}]\!] &= \ldots [\![\texttt{S}]\!] \ldots [\![\texttt{T}]\!] \ldots \\
[\![\texttt{if B then S else T}]\!] &= \ldots [\![\texttt{S}]\!] \ldots [\![\texttt{T}]\!] \ldots \\
[\![\texttt{while B do S}]\!] &= \ldots [\![\texttt{S}]\!] \ldots
\end{aligned}$$

with no occurrences of `S` and `T` in the right-hand sides except as arguments to $[\![\ ]\!]$.

An evaluation function for arithmetic expressions

$$\texttt{eval} : \underbrace{\texttt{AExp}}_{\text{syntax}} \to \underbrace{(\texttt{String} \to \mathbb{Z}) \to \mathbb{Z}}_{\text{semantics}}$$

satisfying equations such as

$$\texttt{eval (AExp.add } e_1\ e_2\texttt{) env = eval } e_1 \texttt{ env + eval } e_2 \texttt{ env}$$

is a denotational semantics, because the semantics of $\texttt{AExp.add } e_1\ e_2$ is defined in terms of the semantics of e_1 and of e_2.

Denotational semantics are a natural match for arithmetic expressions but also functional programs. Now we want a convenient denotational semantics for imperative programs. Because of `while` loops, which are not guaranteed to terminate, we need to develop some additional mathematical notions to reach the point where we can formulate the desired semantics.

11.2 A Relational Denotational Semantics

Denotational semantics for deterministic languages are normally given as a function from prestate to poststate, but relations are more general and more convenient to manipulate. We present a relational denotational semantics.

A denotational semantics of a program will be a mathematical object of type $\texttt{Set (State} \times \texttt{State)}$. The relational approach was also used for the big-step semantics, which took the form of a predicate of type $\texttt{State} \to \texttt{State} \to \texttt{Prop}$. These two types are isomorphic, but $\texttt{Set}\ \alpha$, defined as $\alpha \to \texttt{Prop}$, supports many useful operations and notations, such as $\emptyset$, $\cup$, $\cap$, $\in$, and $\{\ldots \mid \ldots\}$ (Section 7.7).

Our semantics is called `denote`, with $[\![\ ]\!]$ as syntactic sugar. We start with the first four equations of the definition, keeping `while` for later:

```
def denote : Stmt → Set (State × State)
  | Stmt.skip             => Id
  | Stmt.assign x a       =>
    {st | Prod.snd st = (Prod.fst st)[x ↦ a (Prod.fst st)]}
  | Stmt.seq S T          => denote S ○ denote T
  | Stmt.ifThenElse B S T =>
    (denote S ⨾ B) ∪ (denote T ⨾ (fun s ↦ ¬ B s))
```

The `skip` statement is interpreted as the identity relation over states—i.e., the set of all tuples of the form `(s, s)`. This captures the desired semantics of `skip`: The poststate is always identical to the prestate.

The semantics of assignment is the set of tuples where the second component reflects the result of the assignment.

The semantics of sequential composition is elegantly expressed as a relational composition $\bigcirc$, which is defined by the equation

$$r_1 \bigcirc r_2 = \{\texttt{ac} \mid \exists \texttt{b}, (\texttt{Prod.fst ac}, \texttt{b}) \in r_1 \wedge (\texttt{b}, \texttt{Prod.snd ac}) \in r_2\}$$

The definition is perhaps easier to relate to when expressed as

$$r_1 \circ r_2 = \{(a, c) \mid \exists b, (a, b) \in r_1 \wedge (b, c) \in r_2\}$$

Unfortunately, Lean cannot process that form, since it requires the left part of a comprehension to be a single variable.

The semantics of an `if–then–else` statement is given as the union of the semantics of the two branches, restricted to include only the tuples whose first component makes the Boolean condition true or false, depending on the branch. The restriction operator is defined by

$$r \lfloor P = \{ab \mid ab \in r \wedge P\ (\mathtt{Prod.fst}\ ab)\}$$

A more intuitive definition would have been

$$r \lfloor P = \{(a, b) \mid (a, b) \in r \wedge P\ a\}$$

but again this is not supported by Lean. Note two degenerate cases: If `P := (fun _ ↦ True)`, then $r \lfloor P = r$, and if `P := (fun _ ↦ False)`, then $r \lfloor P = \emptyset$.

The difficulties arise when we try to define the semantics of `while` loops. We would like to write

```
| Stmt.whileDo B S      =>
  ((denote S ○ denote (Stmt.whileDo B S)) ⌊ B)
  ∪ (Id ⌊ (fun s ↦ ¬ B s))
```

but this is ill-founded due to the recursive call on `Stmt.whileDo B S`. We need something else. What we are looking for on the right-hand side is some term `X` that satisfies the equation

$$X = ((\mathtt{denote}\ S \circ X) \lfloor B) \cup (\mathtt{Id} \lfloor (\mathtt{fun}\ s \mapsto \neg\ B\ s))$$

We are looking for what mathematicians call a fixpoint. The next four sections are concerned with building an operator `lfp` that computes the fixpoint for a given equation. Using `lfp`, we will be able to define the semantics of `while` loops by

```
| Stmt.whileDo B S      =>
  lfp (fun X ↦ ((denote S ○ X) ⌊ B)
    ∪ (Id ⌊ (fun s ↦ ¬ B s)))
```

11.3 Fixpoints

A *fixpoint* (or *fixed point*) of `f` is a solution for `X` in the equation

$$X = f\ X$$

In general, fixpoints may not exist at all for some `f`; for example, if `f := Nat.succ`, then there exists no value `X` such that `X = Nat.succ X`. There may also be several fixpoints; for example, if `f := (fun x ↦ x)`), then any `X` is a solution of `X = (fun x ↦ x) X = X`. Under some conditions on `f`, a unique *least fixpoint* and a unique *greatest fixpoint* are guaranteed to exist.

Consider the following fixpoint equation, where `X : ℕ → Prop`:

```
X = (fun n : ℕ ↦ n = 0 ∨ (∃m : ℕ, n = m + 2 ∧ X m))
```

This equation is the β-reduced variant of an equation with the right format:

```
X = (fun (P : ℕ → Prop) (n : ℕ) ↦ n = 0 ∨ (∃m : ℕ, n = m + 2 ∧ P m)) X
                                  ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
                                                  f
```

A solution is `X := Even`, the predicate that characterizes the even natural numbers. Recall that we proved the inversion rule

```
Even n ↔ n = 0 ∨ (∃m : ℕ, n = m + 2 ∧ Even m)
```

in Section 6.5. It turns out that `Even` is the only fixpoint. In general, the least and greatest fixpoint may be different. Consider the equation

```
X = (fun P ↦ P) X
```

for `X : ℕ → Prop`. The least fixpoint is `fun _ ↦ False` and the greatest fixpoint is `fun _ ↦ True`. By convention, we have `False < True` and thus `(fun _ ↦ False) < (fun _ ↦ True)`. Similarly, `∅ < @Set.univ α` for any inhabited type `α`.

For the semantics of `while` loops, `X` will have type `Set (State × State)` of relations between states, and `f` will correspond to either taking one extra iteration of the loop (if the condition `B` is true) or the identity (if `B` is false).

Which fixpoint should we use for the semantics of `while`? Whereas the greatest fixpoint would also allow cyclic and diverging executions, the least fixpoint allows only finite (but possibly unbounded) executions. Hence we choose the least fixpoint.

11.4 Monotone Functions

We claimed above that the least and greatest fixpoints are guaranteed to exist under some conditions on `f`. It is time to make this more precise. Let `α` and `β` be arbitrary types, each equipped with a partial order `≤`. The function `f : α → β` is *monotone* if `a ≤ b → f a ≤ f b` for all `a`, `b`. The function `f : Set α → Set α` admits least and greatest fixpoints if `f` is monotone.

Many operations on sets (e.g., union `∪`), relations (e.g., composition `○`), and functions (e.g., the identity function `fun x ↦ x`, the constant function `fun _ ↦ k`, composition `∘`) are monotone or preserve monotonicity. Of course, not all functions are monotone. Here is an example of a nonmonotone function `f` on `Set α`, with `⊆` as the partial order:

```
f A = (if A = ∅ then Set.univ else ∅)
```

If `α` is inhabited, we have `∅ ⊆ Set.univ` but `f ∅ = Set.univ ⊈ ∅ = f Set.univ`.

11.5 Complete Lattices

To define `lfp` for sets (including relations), we need two operations: subset `⊆ : Set α → Set α → Prop` and big intersection `⋂ : Set (Set α) → Set α`, which can be defined as

```
⋂ X = {a | ∀A, A ∈ X → a ∈ A}
```

If `X` is a finite set `{A₁, ..., Aₙ}`, then `⋂ X = A₁ ∩ ⋯ ∩ Aₙ`.

We can define `lfp` more generally so that it works not only with sets but with any instance of the algebraic structure called a complete lattice. A *complete lattice* α is an ordered type for which each `Set` α has an infimum, also called *greatest lower bound*. A complete lattice consists of

1. a partial order $\leq$: $\alpha \to \alpha \to$ `Prop` (i.e., a reflexive, antisymmetric, and transitive binary predicate);
2. an operator $\bigsqcap$: `Set` $\alpha \to \alpha$, called the *infimum*.

The $\bigsqcap$ operator satisfies the following two conditions:

1. $\bigsqcap$ `A` is a lower bound of `A`: $\bigsqcap$ `A` $\leq$ `b` for all `b` $\in$ `A`;
2. $\bigsqcap$ `A` is a greatest lower bound: `b` $\leq \bigsqcap$ `A` for all `b` such that $\forall$`a, a` $\in$ `A` $\to$ `b` $\leq$ `a`.

Together, conditions 1 and 2 ensure that $\bigsqcap$ `A` is the unique greatest lower bound.

The lattice operators $\leq$ and $\bigsqcap$ generalize $\subseteq$: `Set` $\alpha \to$ `Set` $\alpha \to$ `Prop` and $\bigcap$: `Set (Set` α`)` $\to$ `Set` α. Be aware that $\bigsqcap$ `A` need not be in `A`. For example, an open interval $]a, b[$ over $\mathbb{R}$ has infimum $a \notin]a, b[$. The infimum of a set is a generalization of the notion of minimal element.

Here are some examples of complete lattices:

- `Set` α with respect to $\subseteq$ and $\bigcap$ for all types α;
- `Prop` with respect to $\to$ and `fun A` $\mapsto$ $\forall$`a` $\in$ `A, a`;
- `ENat` := $\mathbb{N} \cup \{\infty\}$ with respect to $\leq$ and a suitable infimum operator;
- `EReal` := $\mathbb{R} \cup \{-\infty, \infty\}$ with respect to $\leq$ and a suitable infimum operator.

If α is a complete lattice, then $\beta \to \alpha$ is also a complete lattice. If α and β are complete lattices, then $\alpha \times \beta$ is also a complete lattice. In both cases, $\leq$ and $\bigsqcap$ are defined componentwise.

Here are some nonexamples of complete lattices: $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$, and $\mathbb{R}$ with respect to $\leq$. The issue is that there is no greatest element to assign to $\bigsqcap \emptyset$. Another nonexample is `ERat` := $\mathbb{Q} \cup \{-\infty, \infty\}$, because $\bigsqcap$ `{q | 2 < q * q} = sqrt 2` is not in `ERat`.

In Lean, it is natural to represent complete lattices using type classes:

```
class CompleteLattice (α : Type)
  extends PartialOrder α : Type where
  Inf    : Set α → α
  Inf_le : ∀A b, b ∈ A → Inf A ≤ b
  le_Inf : ∀A b, (∀a, a ∈ A → b ≤ a) → b ≤ Inf A
```

The type `Set` α is an instance of the type class:

```
instance Set.CompleteLattice {α : Type} :
  CompleteLattice (Set α) :=
  { @Set.PartialOrder α with
    Inf         := fun X ↦ {a | ∀A, A ∈ X → a ∈ A}
    Inf_le      := by aesop
    le_Inf      := by aesop }
```

11.6 Least Fixpoint

Using complete lattices, we can define the least fixpoint operator:

$$\texttt{lfp f} = \bigsqcap \{\texttt{x} \mid \texttt{f x} \leq \texttt{x}\}$$

In Lean:

```
def lfp {α : Type} [CompleteLattice α] (f : α → α) : α :=
  CompleteLattice.Inf {a | f a ≤ a}
```

The Knaster–Tarski theorem,[1] which we briefly mentioned in Section 6.1, gives us the following properties for any monotone function `f` on a complete lattice:

- `lfp f` is a fixpoint: `lfp f = f (lfp f)`;
- `lfp f` is smaller than any other fixpoint: `X = f X → lfp f ≤ X`.

11.7 A Relational Denotational Semantics, Continued

With `lfp`, we can fulfill our promise and complete the definition of the denotational semantics of WHILE programs:

```
| Stmt.whileDo B S      =>
  lfp (fun X ↦ ((denote S ○ X) ⨾ B)
    ∪ (Id ⨾ (fun s ↦ ¬ B s)))
```

To validate our definition, we can prove the following connection between the denotational and the big-step semantics:

```
theorem denote_Iff_BigStep (S : Stmt) (s t : State) :
  (s, t) ∈ ⟦S⟧ ↔ (S, s) ⟹ t
```

For the proof, we refer to Chapter 11 of *Concrete Semantics: With Isabelle/HOL* [23] or to the demonstration file accompanying this chapter.

11.8 Application to Program Equivalence

Based on the denotational semantics, we introduce a notion of program equivalence. Two programs are equivalent if they have the same semantics:

```
def DenoteEquiv (S₁ S₂ : Stmt) : Prop :=
  ⟦S₁⟧ = ⟦S₂⟧
```

We write `S₁ ~ S₂` as an abbreviation for `DenoteEquiv S₁ S₂`. It is easy to see that `~` is an equivalence relation.

Program equivalence can be used to replace a subprogram in a larger program with another subprogram if they have the same semantics. This is achieved by the following congruence rules:

```
theorem DenoteEquiv.seq_congr {S₁ S₂ T₁ T₂ : Stmt}
    (hS : S₁ ~ S₂) (hT : T₁ ~ T₂) :
  S₁; T₁ ~ S₂; T₂ :=
  by
```

[1] https://en.wikipedia.org/wiki/Knaster-Tarski_theorem

```
    simp [DenoteEquiv, denote] at *
    simp [*]

theorem DenoteEquiv.if_congr {B} {S₁ S₂ T₁ T₂ : Stmt}
    (hS : S₁ ~ S₂) (hT : T₁ ~ T₂) :
  Stmt.ifThenElse B S₁ T₁ ~ Stmt.ifThenElse B S₂ T₂ :=
  by
    simp [DenoteEquiv, denote] at *
    simp [*]

theorem DenoteEquiv.while_congr {B} {S₁ S₂ : Stmt}
    (hS : S₁ ~ S₂) :
  Stmt.whileDo B S₁ ~ Stmt.whileDo B S₂ :=
  by
    simp [DenoteEquiv, denote] at *
    simp [*]
```

A *congruence rule* is a theorem that lifts an equivalence relation over some context (here, `~` over sequential composition, `if-then-else`, and `while`).

Notice how the denotational semantics leads to short proofs by rewriting. This should not be surprising, given that it is designed to be equational and compositional. If we had used the big-step semantics as the basis for program equivalence instead, these proofs would have been much more complicated.

We can now prove simple programs equivalent by equational reasoning:

```
theorem DenoteEquiv.skip_assign_id {x} :
  Stmt.assign x (fun s ↦ s x) ~ Stmt.skip :=
  by simp [DenoteEquiv, denote, Id]

theorem DenoteEquiv.seq_skip_left {S} :
  Stmt.skip; S ~ S :=
  by simp [DenoteEquiv, denote, Id, comp]

theorem DenoteEquiv.seq_skip_right {S} :
  S; Stmt.skip ~ S :=
  by simp [DenoteEquiv, denote, Id, comp]
```

We defined the semantics of `while` using the `lfp` operator, but who knows whether monotonicity—which guarantees the existence of a least fixpoint—is met? To quell such doubts, we prove the following theorem:

```
theorem DenoteEquiv.if_seq_while {B S} :
  Stmt.ifThenElse B (S; Stmt.whileDo B S) Stmt.skip
  ~ Stmt.whileDo B S :=
  by
    simp [DenoteEquiv, denote]
    apply Eq.symm
    apply lfp_eq
    apply Monotone_while_lfp_arg
```

The theorem gives us a convenient way to expand or contract one iteration of a loop. The second `apply` invokes the theorem `lfp_eq : lfp f = f (lfp f)`, stating

that `lfp` is a fixpoint. The last step applies a theorem to convince Lean that `lfp`'s argument is monotone. That theorem's proof is fairly monotonous:

```
theorem Monotone_while_lfp_arg (S B) :
  Monotone (fun X ↦ ⟦S⟧ ○ X ⨆ B ∪ Id ⨆ (fun s ↦ ¬ B s)) :=
  by
    apply Monotone_union
    { apply SorryTheorems.Monotone_restrict
      apply SorryTheorems.Monotone_comp
      { exact Monotone_const _ }
      { exact Monotone_id } }
    { apply SorryTheorems.Monotone_restrict
      exact Monotone_const _ }
```

11.9 A Simpler Approach Based on an Inductive Predicate

Lean's inductive predicates correspond to least fixpoints, but they are built into Lean's logic (the calculus of inductive constructions), without using a fixpoint operator like `lfp`. Most of this section was dedicated to the construction of an `lfp` operator. Could we have used an inductive predicate instead and saved ourselves this work?

The answer is yes. First, recall that the types `Set (State × State)` and `State → State → Prop` are equivalent representations of binary relations in Lean. So we can build the following `Awhile` predicate, which resembles the reflexive transitive closure but stops when the given condition `B` turns false:

```
inductive Awhile (B : State → Prop)
    (r : Set (State × State)) :
  State → State → Prop
  | true {s t u} (hcond : B s) (hbody : (s, t) ∈ r)
      (hrest : Awhile B r t u) :
    Awhile B r s u
  | false {s} (hcond : ¬ B s) :
    Awhile B r s s
```

With this operator, the definition of the definitional semantics becomes

```
def denoteAwhile : Stmt → Set (State × State)
    ⋮
  | Stmt.whileDo b S      =>
    {st | Awhile b (denoteAwhile S) (Prod.fst st)
       (Prod.snd st)}
```

The `Awhile` predicate's introduction rules closely resembles the `while` rules of the big-step operational semantics. Behind their different façades, operational and denotational semantics are not so dissimilar after all.

Part IV

Mathematics

Chapter 12

Logical Foundations of Mathematics

In this chapter, we dive deeper into the logical foundations of Lean. Most of the features described here are especially relevant for defining mathematical objects and proving theorems about them. For even more details, we refer to Carneiro's MSc thesis [5].

12.1 Universes

In dependent type theory, not only all terms have a type, but also all types have types themselves. We have already seen some occurrences of this principle. The PAT principle tells us to view proofs as terms and propositions as types. For example, the theorem

```
@And.intro : ∀a b, a → b → a ∧ b
```

is really a term `@And.intro` of type `∀a b, a → b → a ∧ b`, which in turn has a type:

```
∀a b, a → b → a ∧ b : Prop
```

What is the type of `Prop` then? `Prop` has the same type as virtually all other types we have constructed so far:

```
Prop : Type
```

What is the type of `Type`? The simplest solution would be to let `Type : Type`, but this choice leads to Girard's paradox, the type theory equivalent of Russel's paradox. To avoid inconsistencies, we need a fresh type to contain `Type`, which we call `Type 1`. `Type 1` itself has type `Type 2`, and so on:

```
Type   : Type 1
Type 1 : Type 2
Type 2 : Type 3
       ⋮
```

In fact, `Type` with no argument is an abbreviation for `Type 0`. If we want to incorporate `Prop` in this hierarchy, we can use the syntax `Sort u`, where `Sort 0` is

an alias for `Prop` and `Sort (u + 1)` is an alias for `Type u`. The hierarchy is captured by the following typing judgment:

$$\frac{}{\texttt{C} \vdash \texttt{Sort u} : \texttt{Sort (u + 1)}}\ \textsc{Sort}$$

All of these types containing other types are called *universes*, and the `u` in the expression `Sort u` is a *universe level*. Although universe levels look like terms of type $\mathbb{N}$, they are in fact not even terms.

Instead of using `Type` everywhere, you can make your theorems slightly more general without having to think about universe levels by writing `Type _`. Lean then creates a fresh universe variable. This helps maintain the illusion that we work in a convenient logic where `Type : Type` holds, but without introducing any paradoxes. In practice, `Type 0` is large enough for most of computer science and mathematics.

12.2 The Peculiarities of Prop

Although `Prop` may seem to fit nicely into the universe hierarchy, it differs from the other universes in several respects.

12.2.1 Impredicativity

When constructing a new type from other types (e.g., $\alpha \to \beta$ from α `: Type u` and β `: Type v`), the newly constructed type is more complex than each of its components, and it is natural to put it into the largest universe involved (e.g., $\alpha \to \beta$ `: Type (max u v)`). This is exactly what Lean does. The following typing rule expresses this idea generally for dependent types:

$$\frac{\texttt{C} \vdash \sigma : \texttt{Type u} \qquad \texttt{C, x} : \sigma \vdash \tau\texttt{[x]} : \texttt{Type v}}{\texttt{C} \vdash \texttt{(x} : \sigma\texttt{)} \to \tau\texttt{[x]} : \texttt{Type (max u v)}}\ \textsc{Arrow-Type}$$

This behavior of the `Type` universes is called *predicativity*. In general, predicativity means that an object *must not* be defined in terms of a quantifier ranging over that same object.

However, it is convenient to have `Prop` behave differently. We would like the expression `∀a : Prop, a → a` to be of type `Prop`—it is, after all, a proposition. Unfolding the syntactic sugar for `∀`, this expression is the same as `(a : Prop) → a → a`. If we had `Type u` instead of `Prop`, the above typing rule would yield

```
(a : Type u) → a → a : Type (u + 1)
```

since `Type u : Type (u + 1)` and `max (u + 1) (max u u) = u + 1`. Thus, the universe level is increased by one when typing this expression. To force expressions such as `∀a : Prop, a → a` to be of type `Prop` anyway, we need a special typing rule for `∀` expressions with a `Prop` body:

$$\frac{\texttt{C} \vdash \sigma : \texttt{Sort u} \qquad \texttt{C, x} : \sigma \vdash \tau\texttt{[x]} : \texttt{Prop}}{\texttt{C} \vdash \texttt{(}\forall\texttt{x} : \sigma\texttt{,}\ \tau\texttt{[x])} : \texttt{Prop}}\ \textsc{Arrow-Prop}$$

The rule yields

```
∀a : Prop, a → a : Prop
```

as desired. The two typing rules above can be summarized as the single rule

$$\frac{\texttt{C} \vdash \sigma : \texttt{Sort u} \qquad \texttt{C, x} : \sigma \vdash \tau\texttt{[x]} : \texttt{Sort v}}{\texttt{C} \vdash \texttt{(x} : \sigma\texttt{)} \to \tau\texttt{[x]} : \texttt{Sort (imax u v)}}\ \textsc{Arrow}$$

where `imax u 0 = 0` and `imax u (v + 1) = max u (v + 1)`. This behavior is called *impredicativity* of `Prop`. In general, impredicativity means that an object *may* be defined via a quantifier ranging over itself.

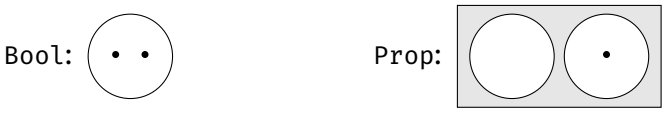
12.2.2 Proof Irrelevance

A second difference between `Prop` and `Type` is *proof irrelevance*. It means that any two proofs of the same proposition `a` are equal:

```
theorem proof_irrel {a : Prop} (h₁ h₂ : a) :
  h₁ = h₂ :=
  by rfl
```

In Lean, this equality is a syntactic equality up to computation, allowing us to use the `rfl` tactic. When viewing a proposition as a type and a proof as an element of that type, proof irrelevance means that a proposition is either an empty type or has exactly one inhabitant. If it is empty, it is false. If it has exactly one inhabitant, it is true. Proof irrelevance is very helpful when reasoning about dependent types.

In Section 6.3, we saw a diagram depicting the interpretation of `Bool` and `Prop` side by side. The diagram did not take proof irrelevance into account and showed multiple proofs for the same proposition. We now know this is not accurate. Here is a revised diagram:



If we mentally connect `Bool`'s two elements (`false` and `true`) with `Prop`'s two types (`False` and `True`), we see that `Prop` is nearly the same as `Bool`, except that propositions of type `Prop` can store proofs, in accordance with the PAT principle. The following table summarizes the situation:

```
                False : Prop
True.intro : True  : Prop
false      : Bool  : Type
true       : Bool  : Type
```

To make proof irrelevance work, Lean must give up the "no confusion" property for inductive predicates. Indeed, various proofs of the same proposition should definitely be "confused." This only concerns inductive predicates (e.g., `Even`), not inductive types in general (e.g., `List α`).

Other systems and logics make different choices. For example, Coq is proof-relevant by default but compatible with proof irrelevance. Homotopy type theory and other constructive or intuitionistic type theories build on data in equality proofs and are therefore incompatible with proof irrelevance.

12.2.3 No Large Elimination

A further difference between `Prop` and `Type` is that `Prop` does not allow *large elimination*: It is generally impossible to extract information from a proof of a proposition and use it in a program (i.e., a value of a type belonging to `Type`). After all, because of proof irrelevance, all proofs of a given proposition are equal, so they cannot carry specific information that would distinguish them.

Imagine we could extract information from proofs inside a program. We could for instance use the `match` construct in function definitions such as the following:

```
-- fails
def unsquare (i : ℤ) (hsq : ∃j, i = j * j) : ℤ :=
  match hsq with
  | Exists.intro j _ => j
```

The `unsquare` function takes a square number `i` and a proof `hsq` that `i` is actually a square number and returns the number `j` before squaring, extracted from the proof. Lean raises the error

```
tactic 'induction' failed, recursor 'Exists.casesOn' can
only eliminate into Prop
```

If it accepted the definition, we could derive `False` as follows. Let

```
hsq₁ := Exists.intro 3 (by linarith)
hsq₂ := Exists.intro (-3) (by linarith)
```

be two proofs of `∃j, (9 : ℤ) = j * j`. Notice that they use different witnesses for `j` (`3` versus `-3`). We then have `unsquare 9 hsq₁ = 3` and `unsquare 9 hsq₂ = -3`. Yet, by proof irrelevance, `hsq₁ = hsq₂`. Hence, `unsquare 9 hsq₂` equals `3`. But we already determined that it equals `-3`. This means `3 = -3`, a contradiction.

An unfortunate consequence of the absence of large elimination is that we cannot perform rule induction by pattern matching and recursion. This kind of induction relies on a "measure"—a function to $\mathbb{N}$ that assigns a size to the arguments. Without large elimination, the measure cannot be defined meaningfully. This explains why we always use the `induction` tactic for rule induction.

As a compromise, Lean allows *small elimination*, which eliminates only into `Prop`—whereas large elimination can eliminate into an arbitrary large universe `Type u`. This means we can use `match` to analyze the structure of a proof, extract existential witnesses, and so on, as long as the `match` expression itself is in a proof. We have seen several examples of this in Section 6.5.

As a further compromise, Lean allows large elimination for *syntactic subsingletons*: types in `Prop` that can be proved in at most one way. For example, `False` has no proof, and all proofs of `a ∧ b` have the form `And.intro _ _`. (Recursively, there might be multiple ways to prove `a` and `b`.) More precisely, a syntactic subsingleton is an inductive definition with at most one constructor whose arguments are either `Prop` or appear as immediate arguments in the result type. When we match `h : a ∧ b` against the pattern `And.intro ha hb`, no information about `h` is leaked.

12.3 The Axiom of Choice

Lean's logic includes the axiom of choice, which makes it possible to obtain an arbitrary element of any nonempty type. Consider the following predicate:

```
inductive Nonempty (α : Sort u) : Prop
  | intro (val : α) : Nonempty α
```

The predicate states that α has at least one element. To prove `Nonempty α`, we must provide an α value to `Nonempty.intro`:

```
theorem Nat.Nonempty :
  Nonempty ℕ :=
  Nonempty.intro 0
```

Since `Nonempty` lives in `Prop`, large elimination is not available, and thus we cannot extract the element that was used in the proof of `Nonempty α`.

In Lean, the axiom of choice takes the form of a function that returns an arbitrary α value given a proof of `Nonempty α`:

```
Classical.choice {α : Sort u} : Nonempty α → α
```

We have no way to know whether the returned element is the same element that was used to prove `Nonempty α`. It will just be an arbitrary element of α.

The constant `Classical.choice` is noncomputable. If we ask Lean for its value using `#reduce` or `#eval`, it will refuse to compute it. In other words, proofs may be terms, but they are not necessarily programs. This is one of the reasons why some logicians prefer to work without the axiom of choice. By contrast, the vast majority of mathematicians have no objections against the axiom.

Unlike proof irrelevance and large and small elimination, the axiom of choice is not built into the Lean kernel; it is only an axiom in the core library, and we are free to work without it. Lean requires us to mark definitions with the `noncomputable` keyword if they use `Classical.choice` to define constants in `Type`.

The following tools rely on `Classical.choice`:

- The function `Classical.choose`, often called Hilbert's choice operator, can help us find a witness of `∃a : α, p a` if we do not care which one. Its companion `Classical.choose_spec` gives a proof that the witness is indeed a witness.

  ```
  Classical.choose : (∃a : α, p a) → α
  Classical.choose_spec : ∀h : (∃a : α, p a), p (Classical.choose h)
  ```

  Intuitively, the choice operator tells us, "Convince me that there exists an element satisfying `p`, and I will give you such an element."

- We can also derive the traditional axiom of choice:

  ```
  Classical.axiomOfChoice (α β : Type) {R : α → β → Prop} :
    (∀x : α, ∃y : β, R x y) → (∃f, ∀x, R x (f x))
  ```

- From the axiom of choice and propositional and functional extensionality (`propext`, `funext`), we can derive the law of excluded middle:

  ```
  Classical.em : ∀a : Prop, a ∨ ¬ a
  ```

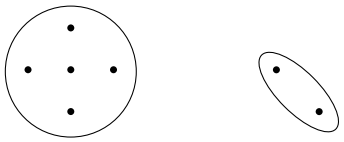
With the law of excluded middle, every proposition is decidable. This means that we can construct proofs based on a case distinction on whether a certain proposition is true.

12.4 Subtypes

Inductive types are a very convenient definitional mechanism when they are applicable, but lots of mathematical constructions do not fit that mold. Lean provides two alternatives to cater for these: subtypes and quotients.

Subtyping is a mechanism to create new types from existing ones. Given a predicate on the elements of a base type, the *subtype* contains only those elements of the base type that satisfy the predicate. More precisely, the subtype contains element–proof pairs that combine an element of the base type and a proof that the predicate is true for that element.

The following diagram depicts a subtype that was created by keeping two of the base type's five elements:



Subtyping is useful for those types that cannot be defined as an inductive type. For example, any attempt at defining the type of finite sets along the lines of

```
inductive Finset (α : Type) : Type
  | empty  : Finset α
  | insert : α → Finset α → Finset α
```

is doomed, because a given set may have multiple representations. For example, `{1, 2}` can be represented in any of the following ways, and more:

```
          Finset.insert 1 (Finset.insert 2 Finset.empty)
          Finset.insert 2 (Finset.insert 1 Finset.empty)
Finset.insert 1 (Finset.insert 1 (Finset.insert 2 Finset.empty))
```

Instead, we can define finite sets as the subtype of (possibly infinite) sets that are finite. In general, subtypes have the syntax

```
{variable : base-type // property-applied-to-variable}
```

We saw an example in Section 4.6, namely, `{i : ℕ // i ≤ n}`, which consists of the natural numbers `i` fulfilling `i ≤ n`, where `n` is fixed in the context. The base type is `ℕ`, and the property is `fun i ↦ i ≤ n`. A less suggestive but perhaps less confusing syntax for the same type is `@Subtype ℕ (fun i ↦ i ≤ n)`. Our motivating example, the type of finite sets over some type `α`, is specified as `{A : Set α // Set.Finite A}`, where `Set.Finite` is `True` if and only if its argument is finite.

12.4.1 First Example: Full Binary Trees

To illustrate subtypes, we will define a type of full binary trees building on the `BTree` type from Section 5.8. In Section 6.6.3, we introduced a predicate `IsFull` that is true if each node of a tree has either zero or two child nodes. Based on this type and this predicate, we can construct a subtype `FullBTree`, containing only full binary trees, as follows:

```
def FullBTree (α : Type) : Type :=
  {t : BTree α // IsFull t}
```

This is syntactic sugar for

```
def FullBTree (α : Type) : Type :=
  @Subtype (BTree α) IsFull
```

where `Subtype` is defined as follows:

```
inductive Subtype {α : Type} (p : α → Prop) : Type
  | mk : (x : α) → p x → Subtype p
```

Elements of `FullBTree` are essentially dependently typed pairs, where the first component is a tree `t` and the second component is a proof that `t` is full:

```
def emptyFullBTree : FullBTree ℕ :=
  Subtype.mk BTree.empty IsFull.empty

def fullBTree6 : FullBTree ℕ :=
  Subtype.mk (BTree.node 6 BTree.empty BTree.empty)
    (by
       apply IsFull.node
       apply IsFull.empty
       apply IsFull.empty
       rfl)
```

Given a value of type `FullBTree`, we can retrieve its two components via `Subtype.val` and `Subtype.property`:

```
#reduce Subtype.val fullBTree6
#check Subtype.property fullBTree6
```

The most appealing aspect of subtypes is that we can *lift* the operations from the base type to the subtype instead of having to build a library from scratch, as long as they preserve the subtype property. We only need to define "wrappers" around constants from the base type. In general, defining such a wrapper around an operation `f` on the base type involves three steps:

1. extract the base type values from the wrapper arguments;
2. invoke `f` on those base type values;
3. encapsulate the result using `Subtype.mk`, together with a proof that the resulting base type value fulfills the subtype property.

Using this procedure, we can lift `BTree` functions to `FullBTree` functions if they preserve the property `IsFull`. For example, to lift the `mirror` operation from the type `BTree → BTree` to the type `FullBTree → FullBTree`, we must

1. extract the `BTree` from the wrapper argument;

2. invoke `mirror` on that `BTree`;
3. encapsulate the result using `Subtype.mk`, together with a proof that the resulting `BTree` fulfills `IsFull`.

For step 3, we must extract the proof of `IsFull` from the argument and use the theorem `IsFull_mirror` from Section 6.6.3. Putting everything together, we get

```
def FullBTree.mirror {α : Type} (t : FullBTree α) :
  FullBTree α :=
  Subtype.mk (LoVe.mirror (Subtype.val t))
    (by
       apply IsFull_mirror
       apply Subtype.property t)
```

The input is an element `t` of the subtype `FullBTree`. We decompose `t` into `Subtype.val t : BTree` and `Subtype.property t : IsFull t`. We use the earlier `mirror` function to reverse the tree component of `t` and use the theorem `IsFull_mirror`, together with the property component of `t`, to show the condition `IsFull (mirror (Subtype.val t))`.

Finally, we build a pair containing the resulting tree and the proof that this tree is full using `Subtype.mk`. The `Subtype.mk` constructor can be seen both as a pair-like constructor and as a cast from `BTree` to `FullBTree`, with a second argument that guarantees that the cast is safe.

For proofs about subtypes, the following theorem is useful:

```
Subtype.eq : Subtype.val ?a = Subtype.val ?b → ?a = ?b
```

It states that two subtype values are equal if their `Subtype.val` components are equal. It is crucial that proofs are irrelevant in Lean, because we would not want to have spurious duplicate values in the subtype, differing only in their proofs. Here is how `Subtype.eq` can be used to prove that the mirror image of the mirror image of a `FullBTree` is the `FullBTree` itself:

```
theorem FullBTree.mirror_mirror {α : Type} (t : FullBTree α) :
  (FullBTree.mirror (FullBTree.mirror t)) = t :=
  by
    apply Subtype.eq
    simp [FullBTree.mirror, LoVe.mirror_mirror]
```

Applying the theorem `Subtype.eq` and unfolding the definition of `full_BTree.mirror` yields the subgoal

```
mirror (mirror (Subtype.val t)) = Subtype.val t
```

which matches the statement of the theorem `mirror_mirror` from Section 5.8.

12.4.2 Second Example: Vectors

As a second example, consider the following definition of vectors:

```
def Vector (α : Type) (n : ℕ) : Type :=
  {xs : List α // List.length xs = n}
```

A vector is defined as a list of a given length. With lists, there is only one type for a list of all lengths. For vectors, we have one dedicated type for every length of a vector. The advantage of this scheme is that some operations, such as addition and scalar product of vectors, require that two involved vectors have the same length. We saw a less practical definition of vectors in Section 5.10.

Vectors can be built from lists using `Subtype.mk`:

```
def vector123 : Vector ℤ 3 :=
  Subtype.mk [1, 2, 3] (by rfl)
```

Basic operations on vectors can be defined by decomposing them with `Subtype.val` and `Subtype.property`, manipulating the underlying lists, and then recomposing them with `Subtype.mk`. For example, we can define the componentwise negation of an integer vector as follows:

```
def Vector.neg {n : ℕ} (v : Vector ℤ n) : Vector ℤ n :=
  Subtype.mk (List.map Int.neg (Subtype.val v))
    (by
       rw [List.length_map]
       exact Subtype.property v)
```

We use the function `List.map` to negate every entry of the underlying list and the theorem `List.length_map` to show that this operation does not change the list's length.

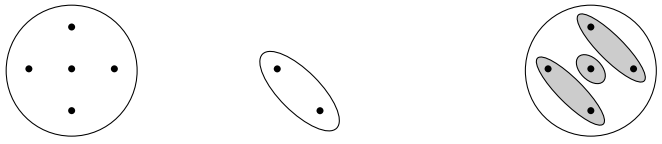
Using `Subtype.eq`, we can prove the following theorem about `Vector.neg`:

```
theorem Vector.neg_neg (n : ℕ) (v : Vector ℤ n) :
  Vector.neg (Vector.neg v) = v :=
  by
    apply Subtype.eq
    simp [Vector.neg]
```

The application of `Subtype.eq` reduces the goal to showing the corresponding property on the underlying lists. We can then use `simp` to finish the proof.

12.5 Quotient Types

Quotients are a powerful construction in mathematics used to define $\mathbb{Z}$, $\mathbb{R}$, and many other sets. Lean supports *quotient types*, an analogous mechanism on types. Like subtyping, quotienting constructs a new type from an existing type. Unlike a subtype, a quotient type contains all of the elements of the base type, except that some elements that are different in the base type may be considered equal in the quotient type. In mathematical terms, the quotient type is isomorphic to a partition of the base type. The diagram below shows a quotient type built from a three-way partition:



The depicted quotient type has only three elements, represented by grayed ellipses. Each of these elements corresponds to one or more base type elements.

The prerequisites to construct a quotient type are a base type τ and an equivalence relation `R : τ → τ → Prop` specifying which elements of the base type will be considered equal in the quotient. To construct the quotient type, we first need to prove that `R` is an equivalence relation on τ. A type τ equipped with an equivalence relation is called a *setoid*. In Lean, `Setoid` is a type class. We can declare an instance using the command

```
instance τ.Setoid : Setoid τ :=
  { r     := R
    iseqv :=
      { refl  := ...
        symm  := ...
        trans := ... } }
```

where the ellipses stand for missing proofs of the respective properties. In addition, this instance declaration introduces the notation `a ≈ b` for `R a b`. More importantly, we can now use the quotient type `Quotient τ.Setoid`.

Every element `a : τ` belongs to some element in `Quotient τ.Setoid`, given by `Quotient.mk τ.Setoid a`, with

```
Quotient.mk {α : Type} → (s : Setoid α) → α → Quotient s
```

The expression `Quotient.mk τ.Setoid a` is quite a mouthful. Fortunately, Lean lets us abbreviate it to to `⟦a⟧`.

The following axiom guarantees that pairs of elements for which `R` holds are indeed equal in the quotient type:

```
Quotient.sound {a b : τ} : a ≈ b → ⟦a⟧ = ⟦b⟧
```

A theorem states the converse:

```
Quotient.exact {a b : τ} : ⟦a⟧ = ⟦b⟧ → a ≈ b
```

Finally, we can lift functions of type $\tau \to \upsilon$, where υ is some arbitrary type, to `Quotient τ.Setoid → υ` using `Quotient.lift`, which satisfies the following syntactic equality up to computation. Given some `f : τ → υ` such that `h : ∀a b, a ≈ b → f a = f b`, we have

```
Quotient.lift f h ⟦a⟧ = f a
```

for all `a : τ`. The argument `h` is a proof that `f` is *compatible* with $\approx$; in other words, it does not distinguish between $\approx$-equivalent arguments.

12.5.1 First Example: Integers

As an example of a quotient type, we will construct the integers. A convenient approach is to construct a quotient over pairs of natural numbers. The idea is that a pair `(p, n)` of natural numbers represents the integer `p - n`. In this way, we can represent all nonnegative integers `p` by `(p, 0)` and all negative integers `- n` by `(0, n)`. We also get many more representations of the same integers; for example, `(7, 0)`, `(8, 1)`, `(9, 2)`, and `(10, 3)` all represent the integer `7`.

First, we need to register the equivalence relation that we want to use. We want two pairs `(p₁, n₁)` and `(p₂, n₂)` to be equal when `p₁ - n₁` and `p₂ - n₂` yield the same integer. However, the condition `p₁ - n₁ = p₂ - n₂` does not work because subtraction on `ℕ` is ill-behaved (e.g., `0 - 1 = 0`). Instead, we use the condition `p₁ + n₂ = p₂ + n₁`, which relies on addition.

We provide the definition of our equivalence relation, followed by a proof that it is reflexive, symmetric, and transitive:

```
instance Int.Setoid : Setoid (ℕ × ℕ) :=
  { r :=
      fun pn₁ pn₂ : ℕ × ℕ ↦
        Prod.fst pn₁ + Prod.snd pn₂ =
        Prod.fst pn₂ + Prod.snd pn₁
    iseqv :=
      { refl :=
          by
            intro pn
            rfl
        symm :=
          by
            intro pn₁ pn₂ h
            rw [h]
        trans :=
          by
            intro pn₁ pn₂ pn₃ h₁₂ h₂₃
            linarith } }
```

We can now write `≈` for the equivalence relation:

```
theorem Int.Setoid_Iff (pn₁ pn₂ : ℕ × ℕ) :
  pn₁ ≈ pn₂ ↔
  Prod.fst pn₁ + Prod.snd pn₂ = Prod.fst pn₂ + Prod.snd pn₁ :=
  by rfl
```

Using the name `int.rel` of our `Setoid` instance that we registered above, we can then define the integers as

```
def Int : Type :=
  Quotient Int.Setoid
```

We can define the integer zero as

```
def Int.zero : Int :=
  ⟦(0, 0)⟧
```

In fact, any term of the form `⟦(m, m)⟧` represents zero:

```
theorem Int.zero_Eq (m : ℕ) :
  Int.zero = ⟦(m, m)⟧ :=
  by
    rw [Int.zero]
    apply Quotient.sound
    rw [Int.Setoid_Iff]
    simp
```

Next, we define addition on our new integers. To define functions on a quotient type, we cannot simply define them by pattern matching, as for inductive types. Instead, we define the function on the base type first and then *lift* the definition to the quotient. To achieve this, we must prove that the definition of the function `f` does not depend on the chosen representative of an equivalence class (i.e., `a ≈ b → f a = f b`). The functions `Quotient.lift` (for unary functions) and `Quotient.lift₂` (for binary functions) can be used to lift functions in this way.

Addition can be defined as adding the pairs of natural numbers componentwise. We then need to provide a proof that this can be lifted to a function on the quotient by showing that `pn₁ ≈ pn₁'` and `pn₂ ≈ pn₂'` imply

```
  ⟦(prod.fst pn₁  + prod.fst pn₂  , prod.snd pn₁  + prod.snd pn₂  ))⟧
= ⟦(prod.fst pn₁' + prod.fst pn₂', prod.snd pn₁' + prod.snd pn₂')⟧
```

Formally:

```
def Int.add : Int → Int → Int :=
  Quotient.lift₂
    (fun pn₁ pn₂ : ℕ × ℕ ↦
       ⟦(Prod.fst pn₁ + Prod.fst pn₂,
         Prod.snd pn₁ + Prod.snd pn₂)⟧)
    (by
       intro pn₁ pn₂ pn₁' pn₂' h₁ h₂
       apply Quotient.sound
       rw [Int.Setoid_Iff] at *
       linarith)
```

The resulting function `Int.add` has the intended behavior:

```
theorem Int.add_Eq (p₁ n₁ p₂ n₂ : ℕ) :
  Int.add ⟦(p₁, n₁)⟧ ⟦(p₂, n₂)⟧ = ⟦(p₁ + p₂, n₁ + n₂)⟧ :=
  by rfl
```

It would be very convenient if Lean let us enter this theorem as the definition of `Int.add` in the first place, presumably with the following syntax:

```
-- fails
def Int.add : Int → Int → Int
  | ⟦(p₁, n₁)⟧, ⟦(p₂, n₂)⟧ => ⟦(p₁ + p₂, n₁ + n₂)⟧
```

This would be a nice and intuitive syntax, but without a proof that the definition is compatible with ≈, we could define nonsensical functions and use them to derive `False`. For example, we could define

```
-- fails
def Int.fst : Int → ℕ
  | ⟦(p, n)⟧ => p
```

Notice that `Int.fst ⟦(0, 0)⟧ = 0` and `Int.fst ⟦(1, 1)⟧ = 1`. However, since `⟦(0, 0)⟧ = ⟦(1, 1)⟧`, we get `0 = 1`, a contradiction.

We can use the characteristic theorem `Int.add_Eq` to prove other theorems about `Int.add`, such as

```
theorem Int.add_zero (i : Int) :
```

```
  Int.add Int.zero i = i :=
  by
    induction i using Quotient.inductionOn with
    | h pn =>
      cases pn with
      | mk p n => simp [Int.zero, Int.add_Eq]
```

We invoke the `induction` tactic with `Quotient.inductionOn` as the induction principle to perform a case distinction on `i`, replacing `i` by ⟦`pn`⟧, where `pn` is an arbitrary value of the base type $\mathbb{N} \times \mathbb{N}$. Then we perform a case distinction on `pn`, obtaining a pair `(p, n)`. Finally, we simplify the goal, using the definition of `Int.zero` and the characteristic equation of `Int.add`.

12.5.2 Second Example: Unordered Pairs

Unordered pairs are pairs for which no distinction is made between the first and second components. They are usually written {*a*, *b*}. We will introduce the type `UPair` α of unordered pairs over α as the quotient of pairs `(a, b)` with respect to the relation "contains the same elements as":

```
instance UPair.Setoid (α : Type) : Setoid (α × α) :=
{ r :=
    fun ab₁ ab₂ : α × α ↦
      ({Prod.fst ab₁, Prod.snd ab₁} : Set α) =
      ({Prod.fst ab₂, Prod.snd ab₂} : Set α)
  iseqv :=
    { refl  := by simp
      symm  := by aesop
      trans := by aesop } }

theorem UPair.Setoid_Iff {α : Type} (ab₁ ab₂ : α × α) :
  ab₁ ≈ ab₂ ↔
  ({Prod.fst ab₁, Prod.snd ab₁} : Set α) =
  ({Prod.fst ab₂, Prod.snd ab₂} : Set α) :=
  by rfl

def UPair (α : Type) : Type :=
  Quotient (UPair.Setoid α)
```

It is easy to prove that our pairs are really unordered:

```
theorem UPair.mk_symm {α : Type} (a b : α) :
  (⟦(a, b)⟧ : UPair α) = ⟦(b, a)⟧ :=
  by
    apply Quotient.sound
    rw [UPair.Setoid_Iff]
    simp [Set.unordered_pair_comm]
```

Another representation of unordered pairs is as sets of cardinality 1 or 2. The following operation converts `UPair` α values to that representation:

```
def Set_of_UPair {α : Type} : UPair α → Set α :=
  Quotient.lift (fun ab : α × α ↦ {Prod.fst ab, Prod.snd ab})
```

```
    (by
      intro ab₁ ab₂ h
      rw [UPair.Setoid_Iff] at *
      exact h)
```

12.5.3 Alternative Definitions via Normalization and Subtyping

Each element of a quotient type correspond to a class of $\approx$-equivalent elements of the base type. If there exists a systematic way to obtain a canonical representative for each $\approx$-equivalence class, we can use a subtype instead of a quotient, keeping only the canonical representatives and filtering out the other elements.

Consider the quotient type `Int` of integers constructed above. We observed that `(7, 0)`, `(8, 1)`, `(9, 2)`, and `(10, 3)` all represent the integer `7`, but intuitively `(7, 0)` seems preferable to the others. We will say that a pair `(p, n)` is *canonical* if `p` or `n` is `0`:

```
inductive Int.IsCanonical : ℕ × ℕ → Prop
  | nonpos {n : ℕ} : Int.IsCanonical (0, n)
  | nonneg {p : ℕ} : Int.IsCanonical (p, 0)
```

The integers then consist of the canonical pairs of natural numbers:

```
def Int : Type :=
  {pn : ℕ × ℕ // Int.IsCanonical pn}
```

Clearly, each integer can be represented in one and only one way. Operations on integers such as addition and multiplication must then provide canonical results. Fortunately, normalizing pairs of natural numbers is easy:

```
def Int.normalize : ℕ × ℕ → ℕ × ℕ
  | (p, n) => if p ≥ n then (p - n, 0) else (0, n - p)

theorem Int.IsCanonical_normalize (pn : ℕ × ℕ) :
  Int.IsCanonical (Int.normalize pn)
```

For unordered pairs, there is no obvious normal form, except to always put the smaller element first (or last). This requires a linear order $\leq$ on α:

```
def UPair.IsCanonical {α : Type} [LinearOrder α] :
  α × α → Prop
  | (a, b) => a ≤ b

def UPair (α : Type) [LinearOrder α] : Type :=
  {ab : α × α // UPair.IsCanonical ab}
```

Returning to the `Int.IsCanonical` predicate, we observe that there are two proofs that `(0, 0)` is canonical, using either `Int.IsCanonical.nonpos` or `Int.IsCanonical.nonneg`. This is not an issue because by proof irrelevance these proofs must be equal.

12.6 Summary of New Lean Constructs

Declaration

`noncomputable`	prefixes noncomputable declarations

Constants

`Classical.choice`	function that returns an arbitrary element of a nonempty type
`Classical.choose`	function that returns a witness given a proof of an existential
`Quotient`	function that creates a quotient type of a given setoid instance
`Quotient.lift`	function that lifts a unary function to a quotient type
`Quotient.lift₂`	function that lifts a binary function to a quotient type
`Setoid`	type class for a type with an equivalence relation on it
`Sort u`	universe at level `u`
`Subtype.mk`	function that constructs a subtype value
`Subtype.property`	function that extracts the property from a subtype value
`Subtype.val`	function that extracts the base value from a subtype value

Notations

`{x : α // P[x]}`	subtype of all `x` in α fulfilling `P[x]`
$\approx$	equivalence relation on a setoid (used for quotienting)
`Prop`	abbreviation for `Sort 0`
`Type u`	abbreviation for `Sort (u + 1)`

Theorems

`Classical.axiomOfChoice`	traditional axiom of choice
`Classical.choose_spec`	characteristic property of `Classical.choose`
`Quotient.exact`	equality on quotient type implies $\approx$ on base type
`Quotient.inductionOn`	induction principle on a quotient type value
`Quotient.sound`	$\approx$ on base type implies equality on quotient type
`Subtype.eq`	equality on base type implies equality of subtype

Chapter 13

Basic Mathematical Structures

In this chapter, we introduce definitions and proofs about basic mathematical structures such as groups, fields, and linear orders.

13.1 Type Classes over a Single Binary Operator

In mathematics, a *group* is a set G with a binary operator $\cdot : G \times G \to G$ fulfilling the following properties, called group axioms:

- associativity: for all a, b, $c \in G$, we have $(a \cdot b) \cdot c = a \cdot (b \cdot c)$;
- identity element: there exists an element $e \in G$ such that for all $a \in G$, we have $e \cdot a = a$;
- inverse element: for each $a \in G$, there exists an inverse element denoted a^{-1} such that $a^{-1} \cdot a = e$.

In Lean, a type class for groups could be defined as follows:

```
class Group (α : Type) where
  mul          : α → α → α
  one          : α
  inv          : α → α
  mul_assoc    : ∀a b c, mul (mul a b) c = mul a (mul b c)
  one_mul      : ∀a, mul one a = a
  mul_left_inv : ∀a, mul (inv a) a = one
```

However, this is not the official definition. Groups are part of a larger hierarchy of algebraic structures.

Group operations can be written multiplicatively (with operator $*$, identity element 1, and inverse element a^{-1}) or additively (with operator +, identity element 0, and inverse element $-a$). This is why Lean offers two type classes for groups: the multiplicative `Group` and the additive `AddGroup`. They are essentially the same but use different names for their constants and properties.

Any type that satisfies the group axioms can be registered as a `Group` or `AddGroup`. To illustrate this, we will define the type `Int2` of integers modulo 2, also known as $\mathbb{Z}/2\mathbb{Z}$ or $\mathbb{Z}_2$, and register it as an `AddGroup`. The type `Int2` has two elements:

```
inductive Int2 : Type
  | zero
```

```
  | one
```

Addition is defined as follows:

```
def Int2.add : Int2 → Int2 → Int2
  | Int2.zero, a         => a
  | Int2.one,  Int2.zero => Int2.one
  | Int2.one,  Int2.one  => Int2.zero
```

To instantiate `AddGroup`, we need to provide the following constants and properties:

```
              add : α → α → α
             zero : α
              neg : α → α
       add_assoc : ∀a b c, add (add a b) c = add a (add b c)
        zero_add : ∀a, add zero a = a
        add_zero : ∀a, add a zero = a
   add_left_neg : ∀a, add (neg a) a = zero
```

The constants `AddGroup.add`, `AddGroup.zero`, and `AddGroup.neg` correspond to the binary operator, the identity element, and the inverse. The properties `AddGroup.add_assoc`, `AddGroup.zero_add`, and `AddGroup.add_left_neg` correspond to the three group axioms. For technical reasons, we must also prove the redundant property `AddGroup.add_zero`.

The type `Int2` can be registered as a group as follows:

```
instance Int2.AddGroup : AddGroup Int2 :=
  { add          := Int2.add
    zero         := Int2.zero
    neg          := fun a ↦ a
    add_assoc    :=
      by
        intro a b c
        cases a <;>
          cases b <;>
          cases c <;>
          rfl
    zero_add     :=
      by
        intro a
        cases a <;>
          rfl
    add_zero     :=
      by
        intro a
        cases a <;>
          rfl
    add_left_neg :=
      by
        intro a
        cases a <;>
```

```
            rfl }
```

Thanks to this instance, we can now write `0`, `+`, and `-`:

```
#reduce Int2.one + 0 - 0 - Int2.one
```

The algebraic hierarchy contains further type classes with one binary operator. The main ones are listed below:

Type class	Properties	Examples
`Semigroup`	associativity of `*`	$\mathbb{R}$, $\mathbb{Q}$, $\mathbb{Z}$, $\mathbb{N}$
`Monoid`	`Semigroup` with unit `1`	$\mathbb{R}$, $\mathbb{Q}$, $\mathbb{Z}$, $\mathbb{N}$
`LeftCancelSemigroup`	`Semigroup` with `c * a = c * b → a = b`	
`RightCancelSemigroup`	`Semigroup` with `a * c = b * c → a = b`	
`Group`	`Monoid` with inverse $^{-1}$	

For most of these structures, commutative versions (where $a \cdot b = b \cdot a$ for all elements a, b) are available: `CommSemigroup`, `CommMonoid`, `CommGroup`. They also all have additive counterparts prefixed with `Add`:

Type class	Properties	Examples
`AddSemigroup`	associativity of `+`	$\mathbb{R}$, $\mathbb{Q}$, $\mathbb{Z}$, $\mathbb{N}$
`AddMonoid`	`AddSemigroup` with unit `0`	$\mathbb{R}$, $\mathbb{Q}$, $\mathbb{Z}$, $\mathbb{N}$
`AddLeftCancelSemigroup`	`AddSemigroup` with `c + a = c + b → a = b`	$\mathbb{R}$, $\mathbb{Q}$, $\mathbb{Z}$, $\mathbb{N}$
`AddRightCancelSemigroup`	`AddSemigroup` with `a + c = b + c → a = b`	$\mathbb{R}$, $\mathbb{Q}$, $\mathbb{Z}$, $\mathbb{N}$
`AddGroup`	`AddMonoid` with inverse `-`	$\mathbb{R}$, $\mathbb{Q}$, $\mathbb{Z}$

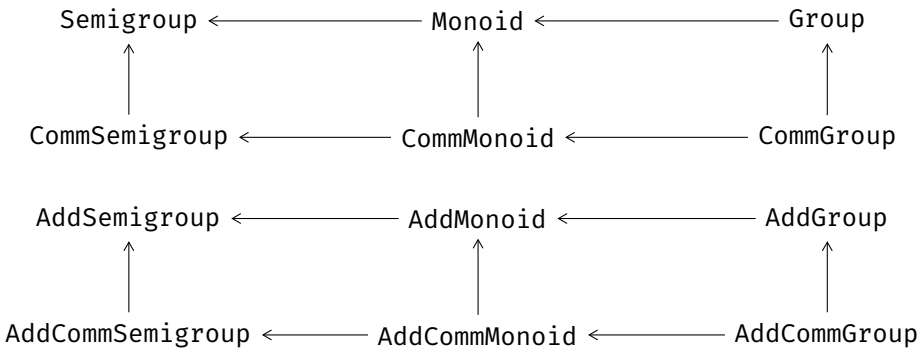
Although the additive type classes are merely copies of their multiplicative counterparts, they are crucial when constructing algebraic structures with more than one binary operator such as rings and fields. To avoid duplicating all theorems and definitions based on the multiplicative type classes, the copying process is automated via metaprograms.

An example instance of `AddMonoid` is the type `List α` with the empty list `[]` as zero and the append operator `++` as addition:

```
instance List.AddMonoid {α : Type} : AddMonoid (List α) :=
  { zero      := []
    add       := fun xs ys ↦ xs ++ ys
    add_assoc := List.append_assoc
    zero_add  := List.nil_append
    add_zero  := List.append_nil }
```

We could continue and register `List α` with `[]` and `++` as an `AddLeftCancelSemigroup` and an `AddRightCancelSemigroup`.

The graph below illustrates the relationships between some of these type classes. An arrow from *X* to *Y* means "*X* inherits all constants and properties from *Y*."



13.2 Type Classes over Two Binary Operators

The additive and multiplicative structures are amalgamated to form more complex structures over two binary operators. One of these is field. A *field* F is defined by the following properties:

- F forms a commutative group under an operator +, called addition, with identity element 0.
- $F \setminus \{0\}$ forms a commutative group under an operator $*$, called multiplication.
- Multiplication distributes over addition—i.e., $a * (b + c) = a * b + a * c$ for all $a, b, c \in F$.

By running `#print Field`, we can display all constants and properties required by `Field`. Again, the type class includes some redundant properties due to its construction.

We will now show that `Int2` is a field by instantiating the `Field` type class with it. First, we must define multiplication on `Int2`:

```
def Int2.mul : Int2 → Int2 → Int2
  | Int2.one,  a => a
  | Int2.zero, _ => Int2.zero
```

To declare `Int2` as a field, we can reuse the instance `Int2.AddGroup` that we defined above using the syntax `Int2.AddGroup with`. We can prove the remaining properties as follows:

```
instance Int2.Field : Field Int2 :=
  { Int2.AddGroup with
    one            := Int2.one
    mul            := Int2.mul
    inv            := fun a ↦ a
    add_comm       :=
      by
        intro a b
        cases a <;>
          cases b <;>
          rfl
    exists_pair_ne :=
      by
        apply Exists.intro Int2.zero
```

```
    apply Exists.intro Int2.one
    simp
zero_mul       :=
  by
    intro a
    rfl
mul_zero       :=
  by
    intro a
    cases a <;>
      rfl
one_mul        :=
  by
    intro a
    rfl
mul_one        :=
  by
    intro a
    cases a <;>
      rfl
mul_inv_cancel :=
  by
    intro a h
    cases a
    { apply False.elim
      apply h
      rfl }
    { rfl }
inv_zero       := by rfl
mul_assoc      :=
  by
    intro a b c
    cases a <;>
      cases b <;>
      cases c <;>
      rfl
mul_comm       :=
  by
    intro a b
    cases a <;>
      cases b <;>
      rfl
left_distrib   :=
  by
    intro a b c
    cases a <;>
      cases b <;>
      rfl
right_distrib  :=
```

```
  by
    intro a b c
    cases a <;>
      cases b <;>
      cases c <;>
      rfl }
```

With this declaration in place, we can now use the notations `1`, `*`, `/`, and more:

```
#reduce (1 : Int2) * 0 / (0 - 1)
```

This command prints `Int2.zero`. The type annotation `: Int2` is necessary here to tell Lean that we want to calculate in `Int2` and not in $\mathbb{N}$, the default. We can even use arbitrary numerals in `Int2`. For example, the numeral `3` is interpreted as `1 + 1 + 1`, which is the same as `1` in `Int2`:

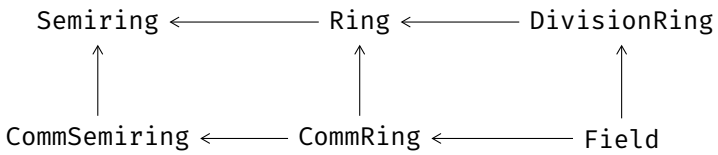
```
#reduce (3 : Int2)
```

This command prints `Int2.one`.

Besides `Field`, there are many more type classes for structures with two binary operators. These are the main ones:

Type class	Properties	Examples
`Semiring`	`Monoid` and `AddCommMonoid` with distributivity	$\mathbb{R}$, $\mathbb{Q}$, $\mathbb{Z}$, $\mathbb{N}$
`CommSemiring`	`Semiring` with commutativity of `*`	$\mathbb{R}$, $\mathbb{Q}$, $\mathbb{Z}$, $\mathbb{N}$
`Ring`	`Monoid` and `AddCommGroup` with distributivity	$\mathbb{R}$, $\mathbb{Q}$, $\mathbb{Z}$
`CommRing`	`Ring` with commutativity of `*`	$\mathbb{R}$, $\mathbb{Q}$, $\mathbb{Z}$
`DivisionRing`	`Ring` with multiplicative inverse	$\mathbb{R}$, $\mathbb{Q}$
`Field`	`DivisionRing` with commutativity of `*`	$\mathbb{R}$, $\mathbb{Q}$

The graph below illustrates the relationships between these type classes.



The hierarchy between `Ring` and `Field` is more complex than depicted and includes type classes for domains, integral domains, and Euclidean rings.

The `Field` type class requires the property `∀a, a / 0 = 0`. This is simply a convention to make division a total function. Mathematicians would regard division as a partial function. There is no harm in totalizing a partial function in this way.

Once we have instantiated the type classes with types of interest, we can use the `ring` tactic to normalize terms containing operators on those types. For example:

```
theorem ring_example (a b : Int2) :
  (a + b) ^ 3 = a ^ 3 + 3 * a ^ 2 * b + 3 * a * b ^ 2 + b ^ 3 :=
  by ring
```

This tactic is available for any type that is declared as a field or more generally as a commutative semiring.

13.3 Coercions

When combining numbers from $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$, and $\mathbb{R}$ in the same theorems, we may want to cast from one type to another. For example, given a natural number, we may need to convert it to an integer. Consider the following theorem and notice the type of the arguments of multiplication:

```
theorem neg_mul_neg_Nat (n : ℕ) (z : ℤ) :
  (- z) * (- n) = z * n :=
  by simp
```

Surprisingly, this statement does not lead to an error, although negation `- n` is not defined on `n : ℕ`, and multiplication of `z : ℤ` and `n : ℕ` is not defined.

The diagnosis command `#check neg_mul_neg_Nat` tells us what happened:

```
neg_mul_neg_Nat : ∀ (n : ℕ) (z : ℤ), -z * -↑n = z * ↑n
```

Lean has a mechanism to introduce coercions, denoted by ↑ or `coe`, when necessary. This coercion operator can be set up to provide implicit conversions between arbitrary types. Many coercions are already in place, including these:

- `coe : ℕ → α` casts $\mathbb{N}$ to another semiring α;
- `coe : ℤ → α` casts $\mathbb{Z}$ to another ring α;
- `coe : ℚ → α` casts $\mathbb{Q}$ to another division ring α.

We can provide type annotations to document our intentions or help Lean figure out where to place the coercions, as in the following example:

```
theorem neg_Nat_mul_neg (n : ℕ) (z : ℤ) :
  (- n : ℤ) * (- z) = n * z :=
  by simp
```

In proofs involving coercions, the `norm_cast` tactic can be convenient. It helps with goals such as `⊢ m n : ℕ, h : ↑m = ↑n ⊢ m = n` in

```
theorem Eq_coe_int_imp_Eq_Nat (m n : ℕ)
    (h : (m : ℤ) = (n : ℤ)) :
  m = n :=
  by norm_cast at h
```

Similarly, it helps with the goal `⊢ m n : ℕ ⊢ ↑m + ↑n = ↑(m + n)` in

```
theorem Nat_coe_Int_add_eq_add_Nat_coe_Int (m n : ℕ) :
  (m : ℤ) + (n : ℤ) = ((m + n : ℕ) : ℤ) :=
  by norm_cast
```

The `norm_cast` tactic relies on theorems such as the following:

```
Nat.cast_add : ∀a b : ℕ, ↑(a + b) = ↑a + ↑b
Int.cast_add : ∀a b : ℤ, ↑(a + b) = ↑a + ↑b
Rat.cast_add : ∀a b : ℚ, ↑(a + b) = ↑a + ↑b
```

13.4 Normalization Tactics

The algebraic tactic `ring` and the coercion tactic `norm_cast` work by normalization: They rewrite expressions in the hope that they become syntactically equal, at

which point equality is trivial to prove. Like `rw` and `simp`, they produce a subgoal when they make some progress but do not fully prove the goal.

The optional *`position`* argument is as for the rewriting tactics (Section 3.5).

ring

```
ring [at position]
```

The `ring` tactic proves equalities over commutative rings and semirings (such as $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$, and $\mathbb{R}$) by normalizing expressions and syntactically comparing the result.

norm_cast

```
norm_cast [at position]
```

The `norm_cast` tactic moves coercions towards the inside of expressions, as a form of simplification.

13.5 Lists, Multisets, and Finite Sets

We have seen many examples of how lists can be used in previous chapters. But when making a new definition or stating a new theorem, we should also reflect on alternatives such as multisets and finite sets.

Consider the following definition, based on the binary trees we introduced in Section 5.8:

```
def List.elems : BTree ℕ → List ℕ
  | BTree.empty      => []
  | BTree.node a l r => a :: List.elems l ++ List.elems r
```

This function returns a list of all the elements occurring in the tree. It traverses the tree depth first, from left to right. But for some applications, we might not care about the order of the elements.

This is where multisets come into play. For multisets, we have `{3, 2, 1, 2} = {1, 2, 2, 3}`, whereas the two lists `[3, 2, 1, 2]` and `[1, 2, 2, 3]` are different. Multisets are defined as the quotient type over lists up to reordering. We can redo the above definition using multisets as follows:

```
def Multiset.elems : BTree ℕ → Multiset ℕ
  | BTree.empty      => ∅
  | BTree.node a l r =>
    {a} ∪ Multiset.elems l ∪ Multiset.elems r
```

Using this definition, we can prove that `Multiset.elems t = Multiset.elems (mirror t)`, whereas `List.elems t = List.elems (mirror t)` does not hold in general.

For some applications, we might want to go a step further and ignore not only the order but also how often each element occurs in the tree, distinguishing only between occurrence and nonoccurrence. This is where finite sets, or *finsets*, come into play. On finsets, we have `{3, 2, 1, 2} = {1, 2, 3}`. Finsets are defined as the subtype of multisets that do not contain any repeated elements. (Another

possible definition would have been as the subtype of sets that are finite.) We can redo the definition above using finsets as follows:

```
def Finset.elems : BTree ℕ → Finset ℕ
  | BTree.empty      => ∅
  | BTree.node a l r => {a} ∪ Finset.elems l ∪ Finset.elems r
```

For lists and multisets, Lean offers sum and product operators to add or multiply all of the elements. Below, the first two commands print 9, and the last two print 24:

```
#eval List.sum [2, 3, 4]
#eval Multiset.sum ({2, 3, 4} : Multiset ℕ)

#eval List.prod [2, 3, 4]
#eval Multiset.prod ({2, 3, 4} : Multiset ℕ)
```

These operators require the type of the elements to be declared as an instance of `AddMonoid` for sum or of `Monoid` for product. The `Multiset` and `Finset` versions also require an instance declaration for `AddCommMonoid` or `CommMonoid` because the result cannot depend on the order of adding or multiplying the elements.

13.6 Order Type Classes

Many of the structures introduced above can be ordered. For example, the familiar order on the natural numbers can be defined as

```
inductive Nat.le : ℕ → ℕ → Prop
  | refl : ∀a : ℕ, Nat.le a a
  | step : ∀a b : ℕ, Nat.le a b → Nat.le a (b + 1)
```

This is an example of a linear order. A *linear order* (or *total order*) is a binary relation $\leq$ such that for all a, b, and c, the following properties hold:

- reflexivity: $a \leq a$;
- transitivity: if $a \leq b$ and $b \leq c$, then $a \leq c$;
- antisymmetry: if $a \leq b$ and $b \leq a$, then $a = b$;
- totality: $a \leq b$ or $b \leq a$.

If a relation has the first three properties, it is a *partial order*. An example is the subset relation $\subseteq$ on sets, finite sets, or multisets. If a relation has the first two properties, it is a *preorder*. An example is comparing lists by their lengths.

In Lean, there are type classes for the different kinds of orders: `LinearOrder`, `PartialOrder`, and `Preorder`. The `Preorder` class has one constant and two properties:

```
le : α → α → Prop
le_refl : ∀a : α, le a a
le_trans : ∀a b c : α, le a b → le b c → le a c
```

The `PartialOrder` class has the additional property

```
le_antisymm : ∀a b : α, le a b → le b a → a = b
```

and `LinearOrder` has the additional property

```
le_total : ∀a b : α, le a b ∨ le b a
```

We can declare the preorder on `List α` that compares lists by their lengths as follows:

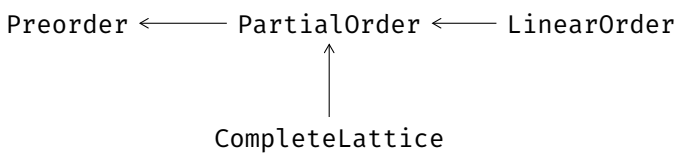
```
instance List.length.Preorder {α : Type} : Preorder (List α) :=
  { le :=
      fun xs ys ↦ List.length xs ≤ List.length ys
    lt :=
      fun xs ys ↦ List.length xs < List.length ys
    le_refl :=
      by
        intro xs
        apply Nat.le_refl
    le_trans :=
      by
        intro xs ys zs
        exact Nat.le_trans
    lt_iff_le_not_le :=
      by
        intro a b
        simp [LT.lt, LE.le]
        intro hlt
        linarith }
```

This type class instance gives access to the infix syntax $\leq$ and to the corresponding relations $\geq$, <, and >:

```
theorem list.length.Preorder_example :
  [1] > [] :=
  by decide
```

The proof features a new tactic: `decide`. It relies on type class inference to prove trivial decidable goals.

Complete lattices, which we discussed in Chapter 11, are formalized as another type class, `CompleteLattice`, which extends `PartialOrder`. In the following diagram, an arrow from *X* to *Y* means "*X* inherits all constants and properties from *Y*."



Finally, Lean provides type classes that combine orders and algebraic structures: `OrderedCancelCommMonoid`, `OrderedCommGroup`, `OrderedSemiring`, `LinearOrderedSemiring`, `LinearOrderedCommRing`, `LinearOrderedField`. All these mathematical structures relate $\leq$ and < with the constants `0`, `1`, `+`, and `*` by monotonicity rules (e.g., `a ≤ b → c ≤ d → a + c ≤ b + d`) and cancellation rules (e.g., `c + a ≤ c + b → a ≤ b`).

13.7 Decision Tactic

decide

The `decide` tactic can be used to prove true decidable goals. Decidability is determined by checking membership of the `Decidable` type class. Unlike `rfl`, `decide` is not limited to proving equalities.

13.8 Summary of New Lean Constructs

Notation

↑	coercion operator `coe`

Tactics

`decide`	proves decidable truths (e.g., a true executable expression)
`norm_cast`	normalizes coercions
`ring`	normalizes ring expressions

Chapter 14

Rational and Real Numbers

We have seen how the natural numbers $\mathbb{N}$ can be defined as an inductive type and how the integers $\mathbb{Z}$ can be defined as a quotient over $\mathbb{N} \times \mathbb{N}$. In this chapter, we review the construction of the rational numbers $\mathbb{Q}$ and the real numbers $\mathbb{R}$. The tools used for these constructions are inductive types, subtypes, and quotients.

The following procedure can be used to construct types that enjoy specific properties:

1. Create a new type that is large enough to represent all elements, but not necessarily in a unique manner.
2. Take the quotient of this representation, equating elements as needed.
3. Define operators on the quotient type by lifting functions from the base type, and prove that they are compatible with the quotient relation.

We used this approach to construct the type $\mathbb{Z}$ in Section 12.5.1. It can be used for $\mathbb{Q}$ and $\mathbb{R}$ as well.

14.1 Rational Numbers

A rational number is a number that can be expressed as a fraction n/d of integers n and d, where $d \neq 0$:

```
structure Fraction where
  num           : ℤ
  denom         : ℤ
  denom_Neq_zero : denom ≠ 0
```

The number n is called the numerator, and the number d is called the denominator. The representation of a rational number as a fraction is not unique. For example, the rationals $1/2$, $2/4$, and $-1/-2$ are all equal. This representation as a fraction will serve as the base type of which we will take the quotient.

Two fractions n_1/d_1 and n_2/d_2 represent the same rational number if the ratio between numerator and denominator are the same: $n_1 * d_2 = n_2 * d_1$. To construct the quotient over the type `Fraction` with respect to this relation, we show that the relation is an equivalence relation. This is achieved by declaring `Fraction` an instance of the `Setoid` type class:

```
namespace Fraction
```

```
instance Setoid : Setoid Fraction :=
  { r :=
      fun a b : Fraction ↦ num a * denom b = num b * denom a
    iseqv :=
      { refl  := by aesop
        symm  := by aesop
        trans :=
          by
            intro a b c heq_ab heq_bc
            apply Int.eq_of_mul_eq_mul_right (denom_Neq_zero b)
            calc
              num a * denom c * denom b
              = num a * denom b * denom c :=
                by ac_rfl
              _ = num b * denom a * denom c :=
                by rw [heq_ab]
              _ = num b * denom c * denom a :=
                by ac_rfl
              _ = num c * denom b * denom a :=
                by rw [heq_bc]
              _ = num c * denom a * denom b :=
                by ac_rfl
            } }

theorem Setoid_Iff (a b : Fraction) :
  a ≈ b ↔ num a * denom b = num b * denom a :=
  by rfl

end Fraction
```

Then we can define the type of rationals as the quotient over this setoid:

```
def Rat : Type :=
  Quotient Fraction.Setoid
```

To define zero, one, addition, multiplication, and other operations, we first define them on the `Fraction` type. To add two fractions, we convert them to a common denominator and add the numerators. The easiest common denominator to use is simply the product of the two denominators:

```
instance Add : Add Fraction :=
  { add := fun a b : Fraction ↦
      { num            := num a * denom b + num b * denom a
        denom          := denom a * denom b
        denom_Neq_zero := by simp [denom_Neq_zero] } }
```

We register these operations as instances of the syntactic type classes such as `Add` to be able to use convenient notation such as `+` on `Fraction`. Similarly, we define zero as `0 := 0/1`, one as `1 := 1/1`, and multiplication as the pairwise multiplication of numerators and denominators.

To lift these operations to the type of rational numbers, `Rat`, we must prove them compatible with ≈:

```
namespace Fraction

@[simp] theorem add_num (a b : Fraction) :
  num (a + b) = num a * denom b + num b * denom a :=
  by rfl

@[simp] theorem add_denom (a b : Fraction) :
  denom (a + b) = denom a * denom b :=
  by rfl

theorem Setoid_add {a a' b b' : Fraction} (ha : a ≈ a')
    (hb : b ≈ b') :
  a + b ≈ a' + b' :=
  by
    simp [Setoid_Iff, add_denom, add_num] at *
    calc
      (num a * denom b + num b * denom a)
          * (denom a' * denom b')
      = num a * denom a' * denom b * denom b'
        + num b * denom b' * denom a * denom a' :=
        by
          simp [add_mul, mul_add]
          ac_rfl
      _ = num a' * denom a * denom b * denom b'
            + num b' * denom b * denom a * denom a' :=
        by simp [*]
      _ = (num a' * denom b' + num b' * denom a')
            * (denom a * denom b) :=
        by
          simp [add_mul, mul_add]
          ac_rfl

end Fraction
```

Then we can use `Quotient.lift`($_2$) to define the `Rat` operations, and we can instantiate the relevant syntactic type classes, such as

```
namespace Rat

instance Add : Add Rat :=
  { add := Quotient.lift₂ (fun a b : Fraction ↦ mk (a + b))
      (by
         intro a b a' b' ha hb
         apply Quotient.sound
         exact Fraction.Setoid_add ha hb) }

end Rat
```

From here, we can proceed and prove all the properties needed to make `Rat` an instance of `Field`.

Alternative Definitions of the Rational Numbers In `mathlib`, another approach is used to define the rationals. The type `Rat` is defined as a subtype of `Fraction`, with the requirement that the denominator is positive and that the numerator and the denominator are coprime (i.e., they have no common divisors except 1 and -1):

```
def Rat.IsCanonical (a : Fraction) : Prop :=
  Fraction.denom a > 0
  ∧ Nat.coprime (Int.natAbs (Fraction.num a))
      (Int.natAbs (Fraction.denom a))

def Rat : Type :=
  {a : Fraction // Rat.IsCanonical a}
```

This is an instance of the general strategy described in Section 12.5.3. With this approach, no quotient is required, computation is more efficient, and more properties are syntactic equalities up to computation. A drawback is that function definitions become more complicated due to the need to normalize fractions.

14.2 Real Numbers

Some sequences of rational numbers seem to converge because the numbers in the sequence get closer and closer to each other and yet do not converge to a rational number. The sequence

$$\begin{aligned} a_0 &= 1 \\ a_1 &= 1.4 \\ a_2 &= 1.41 \\ a_3 &= 1.414 \\ a_4 &= 1.4142 \\ &\vdots \end{aligned}$$

where a_n is the largest number with n digits after the decimal point such that $a_n^2 < 2$, is such a sequence. It seems to converge because each a_n is at most 10^{-n} away from any of the following numbers, but the limit is $\sqrt{2} \notin \mathbb{Q}$. In that sense, the rational numbers are incomplete, and the reals are their *completion*. To construct the reals, we need to fill in the gaps that are revealed by these sequences that seem to converge but do not.

Cauchy sequences capture the notion of a sequence that seems to converge. A sequence $a_0, a_1, \ldots$ is *Cauchy* if for any $\varepsilon > 0$, there exists an $N \in \mathbb{N}$ such that for all $m \geq N$, we have $|a_N - a_m| < \varepsilon$. In other words, no matter how small we choose ε, we can always find a point in the sequence from which all following numbers deviate by less than ε.

We formalize sequences of rational numbers as functions `f : ℕ → ℚ` and denote the absolute value $|\ |$ by `abs`. This yields the following Lean definition of Cauchy sequences:

```
def IsCauchySeq (f : ℕ → ℚ) : Prop :=
  ∀ε > 0, ∃N, ∀m ≥ N, abs (f N - f m) < ε
```

Not every sequence is a Cauchy sequence:

```
theorem id_Not_CauchySeq :
  ¬ IsCauchySeq (fun n : ℕ ↦ (n : ℚ)) :=
  by
    rw [IsCauchySeq]
    intro h
    cases h 1 zero_lt_one with
    | intro i hi =>
      have hi_succi :=
        hi (i + 1) (by simp)
      simp [←sub_sub] at hi_succi
```

We define a type of Cauchy sequences as a subtype:

```
def CauchySeq : Type :=
  {f : ℕ → ℚ // IsCauchySeq f}
```

It will be convenient to have an auxiliary function that extracts the actual sequence from a `CauchySeq`:

```
def seqOf (f : CauchySeq) : ℕ → ℚ :=
  Subtype.val f
```

The basic idea of the construction is to represent the real numbers by Cauchy sequences. Each Cauchy sequence represents the real number that is its limit; for example, the sequence $a_n = 1/n$ represents the real number 0, and the sequence 1, 1.4, 1.41, ... represents the real number $\sqrt{2}$.

Two different Cauchy sequences can represent the same real number; for example, the sequence $a_n = 1/n$ and the constant sequence $b_n = 0$ both represent 0. Therefore, we need to take the quotient over sequences representing the same real number. Two sequences represent the same real number when their difference converges to zero:

```
namespace CauchySeq

instance Setoid : Setoid CauchySeq :=
{ r :=
    fun f g : CauchySeq ↦
      ∀ε > 0, ∃N, ∀m ≥ N, abs (seqOf f m - seqOf g m) < ε
  iseqv :=
    { refl :=
        by
          intro f ε hε
          apply Exists.intro 0
          aesop
      symm :=
        by
          intro f g hfg ε hε
          cases hfg ε hε with
          | intro N hN =>
            apply Exists.intro N
            intro m hm
```

```
            rw [abs_sub_comm]
            apply hN m hm
        trans :=
          by
            intro f g h hfg hgh ε hε
            cases hfg (ε / 2) (half_pos hε) with
            | intro N₁ hN₁ =>
              cases hgh (ε / 2) (half_pos hε) with
              | intro N₂ hN₂ =>
                apply Exists.intro (max N₁ N₂)
                intro m hm
                calc
                  abs (seqOf f m - seqOf h m)
                  ≤ abs (seqOf f m - seqOf g m)
                    + abs (seqOf g m - seqOf h m) :=
                    by apply abs_sub_le
                  _ < ε / 2 + ε / 2 :=
                    add_lt_add (hN₁ m (le_of_max_le_left hm))
                      (hN₂ m (le_of_max_le_right hm))
                  _ = ε :=
                    by simp } }

  theorem Setoid_iff (f g : CauchySeq) :
    f ≈ g ↔
    ∀ε > 0, ∃N, ∀m ≥ N, abs (seqOf f m - seqOf g m) < ε :=
    by rfl

  end CauchySeq
```

Using this `Setoid` instance, we can now define the real numbers:

```
  def Real : Type :=
    Quotient CauchySeq.Setoid
```

Like for the rational numbers, we need to define zero, one, addition, multiplication, and other operators. We define them on `CauchySeq` first and lift them to `Real` afterwards. For the constants `0` and `1`, we can define them simply as constant sequences. Any constant sequence is a Cauchy sequence:

```
  namespace CauchySeq

  def const (q : ℚ) : CauchySeq :=
    Subtype.mk (fun _ : ℕ ↦ q)
      (by
        rw [IsCauchySeq]
        intro ε hε
        aesop)
```

We can declare `Real` instances of the syntactic type classes `Zero` and `One`:

```
  instance Zero : Zero Real :=
    { zero := ⟦CauchySeq.const 0⟧ }
```

```
instance One : One Real :=
  { one := ⟦CauchySeq.const 1⟧ }
```

Defining addition of real numbers requires a little more effort. We define addition on Cauchy sequences by adding the elements of the sequence pairwise:

```
instance Add : Add CauchySeq :=
  { add := fun f g : CauchySeq ↦
      Subtype.mk (fun n : ℕ ↦ seqOf f n + seqOf g n) sorry }
```

This definition requires a proof that the result is a Cauchy sequence, given that `f` and `g` are Cauchy sequences. It is omitted.

Next, we need to show that this addition is compatible with $\approx$:

```
theorem Setoid_add {f f' g g' : CauchySeq} (hf : f ≈ f')
    (hg : g ≈ g') :
  f + g ≈ f' + g' :=
  by
    intro ε₀ hε₀
    simp [Setoid_iff]
    cases hf (ε₀ / 2) (half_pos hε₀) with
    | intro Nf hNf =>
      cases hg (ε₀ / 2) (half_pos hε₀) with
      | intro Ng hNg =>
        apply Exists.intro (max Nf Ng)
        intro m hm
        calc
          abs (seqOf (f + g) m - seqOf (f' + g') m)
          = abs ((seqOf f m + seqOf g m)
            - (seqOf f' m + seqOf g' m)) :=
            by rfl
          _ = abs ((seqOf f m - seqOf f' m)
              + (seqOf g m - seqOf g' m)) :=
            by
              have arg_eq :
                seqOf f m + seqOf g m
                  - (seqOf f' m + seqOf g' m) =
                seqOf f m - seqOf f' m
                  + (seqOf g m - seqOf g' m) :=
                by linarith
              rw [arg_eq]
          _ ≤ abs (seqOf f m - seqOf f' m)
              + abs (seqOf g m - seqOf g' m) :=
            by apply abs_add
          _ < ε₀ / 2 + ε₀ / 2 :=
            add_lt_add (hNf m (le_of_max_le_left hm))
              (hNg m (le_of_max_le_right hm))
          _ = ε₀ :=
            by simp

end CauchySeq
```

To prove that `f + g ≈ f' + g'`, we are given an `ε₀ > 0` and must show that there exists a number `N` such that

$$\forall m,\ m \geq N \to \mathtt{abs}\ (\mathtt{seqOf}\ (f + g)\ m - \mathtt{seqOf}\ (f' + g')\ m) < \varepsilon_0$$

To obtain `N`, we use `f ≈ f'` and `g ≈ g'`. The equivalence `f ≈ f'` gives us for any `ε > 0` a number `Nf` such that `abs (seqOf f m - seqOf f' m) < ε` for all `m ≥ Nf`. The fact `g ≈ g'` gives us a number `Ng` with a similar property. For the calculations to work out in the end, we take the numbers `Nf` and `Ng` for `ε := ε₀ / 2`. Then we choose `N` to be the maximum of `Nf` and `Ng`, so that we get the inequalities for any `m ≥ N`. The `calc` block at the end of the proof establishes that, for all `m ≥ N`,

$$\mathtt{abs}\ (\mathtt{seqOf}\ (f + g)\ m - \mathtt{seqOf}\ (f' + g')\ m) < \varepsilon_0$$

Having shown that addition on Cauchy sequences is compatible with ≈, we can define addition on `Real`:

```
namespace Real

instance Add : Add Real :=
{ add := Quotient.lift₂ (fun a b : CauchySeq ↦ ⟦a + b⟧)
    (by
       intro a b a' b' ha hb
       apply Quotient.sound
       exact CauchySeq.Setoid_add ha hb) }

end Real
```

We could continue like this with multiplication and other operators. In summary, real numbers are defined as a quotient over Cauchy sequences, which are in turn defined as a subtype of $\mathbb{N} \to \mathbb{Q}$.

Alternative Definitions of the Real Numbers In `mathlib`, the construction of the real numbers is essentially as described above. Some definitions are stated in a more general fashion to allow construction of other algebraic structures, such as the *p*-adic numbers [18].

Alternatively, the real numbers can be defined using Dedekind cuts. A number `r : ℝ` is then represented as the set of numbers `x : ℚ` such that `x < r`. Another alternative, which does not depend on $\mathbb{Q}$, is to define $\mathbb{R}$ using binary sequences $\mathbb{N} \to$ `Bool`. The elements of the sequence represent the number's digits. This works particularly well if we only need the real interval $[0, 1]$.

14.3 Final Exhortation

We have now reached the end of this guide. You now know fundamental theory and techniques in interactive theorem proving as well as some application areas. Although we used Lean, your skills should help you with other systems, especially those based on simple or dependent type theory. You should also be able to read many if not most scientific papers in the area.

Even if you do not choose to pursue a career in theorem proving, the authors hope you will bring proof assistants with you and use them when it makes sense,

either because of their high trustworthiness or because of their convenience for keeping track of complex proof goals.

If you continue using Lean, the natural next step would be to familiarize yourself with `mathlib` and its documentation. If you use Lean outside a class context, you will often find yourself looking up definitions and theorems. The `#find` command will surely be useful.[1] And the Lean Zulip chat[2] is the meeting place of Lean users.

14.4 Summary of New Lean Constructs

Command

`#find`	looks up a definition or theorem by pattern matching

[1] https://leanprover-community.github.io/mathlib_docs/commands.html#find
[2] https://leanprover.zulipchat.com/

Bibliography

[1] *The Lean 4 Manual*. 2021. https://leanprover.github.io/lean4/doc/.

[2] J. Avigad, L. de Moura, and J. Roesch. *Programming in Lean*. 2016. https://leanprover.github.io/programming_in_lean/.

[3] H. P. Barendregt, W. Dekkers, and R. Statman. *Lambda Calculus with Types*. Perspectives in logic. Cambridge University Press, 2013.

[4] K. Buzzard, J. Commelin, and P. Massot. Formalising perfectoid spaces. In J. Blanchette and C. Hritcu, editors, *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2020, New Orleans, LA, USA, January 20-21, 2020*, pages 299–312. ACM, 2020.

[5] M. Carneiro. The type theory of Lean. MSc thesis, Carnegie Mellon University, 2019. https://github.com/digama0/lean-type-theory/releases/download/v1.0/main.pdf.

[6] G. Gonthier. Formal proof—the Four-Color Theorem. *Notices AMS*, 55(11):1382–1393, 2008. https://www.ams.org/notices/200811/tx081101382p.pdf.

[7] G. Gonthier, A. Asperti, J. Avigad, Y. Bertot, C. Cohen, F. Garillot, S. L. Roux, A. Mahboubi, R. O'Connor, S. O. Biha, I. Pasca, L. Rideau, A. Solovyev, E. Tassi, and L. Théry. A machine-checked proof of the Odd Order Theorem. In S. Blazy, C. Paulin-Mohring, and D. Pichardie, editors, *ITP 2013*, volume 7998 of *Lecture Notes in Computer Science*, pages 163–179. Springer, 2013. https://hal.inria.fr/hal-00816699/document.

[8] K. Gopinathan and I. Sergey. Certifying certainty and uncertainty in approximate membership query structures. In S. K. Lahiri and C. Wang, editors, *CAV 2020, Part II*, volume 12225 of *Lecture Notes in Computer Science*, pages 279–303. Springer, 2020. https://ilyasergey.net/papers/ceramist-draft.pdf.

[9] R. Gu, Z. Shao, H. Chen, X. N. Wu, J. Kim, V. Sjöberg, and D. Costanzo. CertiKOS: An extensible architecture for building certified concurrent OS kernels. In K. Keeton and T. Roscoe, editors, *OSDI 2016*, pages 653–669. USENIX Association, 2016. https://www.usenix.org/system/files/conference/osdi16/osdi16-gu.pdf.

[10] T. C. Hales, M. Adams, G. Bauer, D. T. Dang, J. Harrison, T. L. Hoang, C. Kaliszyk, V. Magron, S. McLaughlin, T. T. Nguyen, T. Q. Nguyen, T. Nipkow, S. Obua,

J. Pleso, J. Rute, A. Solovyev, A. H. T. Ta, T. N. Tran, D. T. Trieu, J. Urban, K. K. Vu, and R. Zumkeller. A formal proof of the Kepler conjecture. *CoRR*, abs/1501.02155, 2015. http://arxiv.org/abs/1501.02155.

[11] J. Harrison. Formal verification at Intel. In *LICS 2003*, pages 45–54. IEEE Computer Society, 2003. https://ieeexplore.ieee.org/document/1210044.

[12] J. Harrison, J. Urban, and F. Wiedijk. History of interactive theorem proving. In J. H. Siekmann, editor, *Computational Logic*, volume 9 of *Handbook of the History of Logic*, pages 135–214. Elsevier, 2014. https://www.cl.cam.ac.uk/~jrh13/papers/joerg.pdf.

[13] S. K. Jeremy Avigad, Leonardo de Moura and S. Ullrich. *Theorem Proving in Lean 4*. 2021. https://leanprover.github.io/theorem_proving_in_lean4/.

[14] G. Klein, J. Andronick, K. Elphinstone, G. Heiser, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal verification of an operating-system kernel. *Commun. ACM*, 53(6):107–115, 2010. https://www.researchgate.net/profile/Gernot_Heiser/publication/220910193_SeL4_Formal_verification_of_an_OS_kernel/links/09e4150f00292a0329000000/SeL4-Formal-verification-of-an-OS-kernel.pdf.

[15] D. E. Knuth. *The TEXbook*. Addison-Wesley, 1986. http://www.ctex.org/documents/shredder/src/texbook.pdf.

[16] R. Kumar, M. O. Myreen, M. Norrish, and S. Owens. CakeML: A verified implementation of ML. In S. Jagannathan and P. Sewell, editors, *POPL 2014*, pages 179–192. ACM, 2014. https://cakeml.org/popl14.pdf.

[17] X. Leroy. A formally verified compiler back-end. *J. Automated Reasoning*, 43(4):363–446, 2009. https://arxiv.org/pdf/0902.2137.pdf.

[18] R. Y. Lewis. A formal proof of Hensel's lemma over the *p*-adic integers. In A. Mahboubi and M. O. Myreen, editors, *CPP 2019*, pages 15–26. ACM, 2019. https://arxiv.org/pdf/1909.11342.pdf.

[19] J. Limperg and A. H. From. Aesop: White-box best-first proof search for Lean. In B. Pientka and S. Zdancewic, editors, *CPP '23*. ACM, 2023. https://people.compute.dtu.dk/ahfrom/aesop-camera-ready.pdf.

[20] A. Lochbihler. Verifying a compiler for Java threads. In A. D. Gordon, editor, *ESOP 2010*, volume 6012 of *Lecture Notes in Computer Science*, pages 427–447. Springer, 2010. https://link.springer.com/content/pdf/10.1007/978-3-642-11957-6_23.pdf.

[21] The mathlib Community. The Lean mathematical library. In J. Blanchette and C. Hriţcu, editors, *CPP 2020*, pages 367–381. ACM, 2020. https://arxiv.org/pdf/1910.09336.pdf.

[22] R. Nederpelt and H. Geuvers. *Type Theory and Formal Proof: An Introduction*. Cambridge University Press, 2014.

[23] T. Nipkow and G. Klein. *Concrete Semantics: With Isabelle/HOL*. Springer, 2014. http://www.concrete-semantics.org/concrete-semantics.pdf.

[24] B. O'Sullivan, D. Stewart, and J. Goerzen. *Real World Haskell*. O'Reilly, 2008. http://book.realworldhaskell.org/read/.

[25] A. J. Perlis. Epigrams on programming. *SIGPLAN Notices*, 17(9):7–13, 1982. https://dl.acm.org/doi/pdf/10.1145/947955.1083808?casa_token=Z_grmOD_yAYAAAAA:dvMu7thQc-8XmGJCZ1ARQ24_XB1qel3M1jYUmBPAKfuBaZrryxmKiKValMrwpunJ0dv9vhr2kDvMxA.

[26] B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[27] F. van Raamsdonk. *Logical Verification: Course Notes*. 2011. https://www.cs.vu.nl/~jbe248/lv2017/notes.pdf.

[28] D. M. Russinoff. A mechanically checked proof of correctness of the AMD K5 floating point square root microcode. *Formal Methods in System Design*, 14(1):75–125, 1999. https://link.springer.com/content/pdf/10.1023/A:1008669628911.pdf.

[29] C. Watt. Mechanising and verifying the WebAssembly specification. In J. Andronick and A. P. Felty, editors, *CPP 2018*, pages 53–65. ACM, 2018. https://www.cl.cam.ac.uk/~caw77/papers/mechanising-and-verifying-the-webassembly-specification.pdf.