Tim Baanen
Alexander Bentkamp
Jasmin Blanchette
Johannes Hölzl

Logical Verification in Lean

2019 Edition
(November 21, 2019)

lean-forward.github.io/
logical-verification/2019

Contents

Preface

Formal proof assistants are software tools designed to help their users carry out computer-checked proofs in a logical calculus. We usually call them *proof assistants*, or *interactive theorem provers*, but a mischievous student coined the phase "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, already starting with the key notion of "proof." A *formal proof* is a logical argument expressed in a logical formalism. "Formal" is roughly synonymous with "logical" or "logic-based." Logicians—the mathematicians of logics—carried out formal proofs on papers decades before the advent of computers, but nowadays these 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 the 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, not only toy examples or logic puzzles. Formal proofs also helps 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.

Formal proofs can help when we develop a new theory, to find out whether it works and exactly which assumptions are needed. They are useful when generalizing, refactoring, or otherwise modifying existing proof, in much the same way as a compiler helps developing correct programs. They give a high level of trustworthiness that makes it easier to review the proof. In addition, formal proofs can form the basis of verified computational tools (e.g., verified computer algebra).

Success Stories There have been a number of success stories in mathematics and computer science. Three landmark results in the formalization of mathematics have been the proof of the four-color theorem by Gonthier et al. [6], the proof

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

of the odd-order theorem by Gonthier et al. [7], and the proof of the Kepler conjecture by Hales et al. [9]. 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 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 today are computer scientists. A few companies, including AMD [21] and Intel [10], have been using proof assistants to verify their designs. In academia, some of the milestones are the verifications of the operating system kernels seL4 [12] and CertiKOS [8] and the development of the verified compilers CompCert [13] and JinjaThreads [14].

A Selection of Popular Proof Assistants There are dozens of proof assistants in development or use. A list of the main ones follows, classified by logical foundations:

- set theory: Isabelle/ZF, Metamath, Mizar;
- simple type theory (also called higher-order logic): HOL4, HOL Light, Isabelle/HOL;
- dependent type theory: Agda, Coq, Lean, Matita, PVS.

ACL2 is hard to classify. It is based on a Lisp-like first-order logic.

Lean Lean is a new proof assistant developed primarily by Leonardo de Moura (Microsoft Research) since 2012. Its mathematical library, `mathlib`, is developed under the leadership of Jeremy Avigad (Carnegie Mellon University).

We will use version 3.4.2, which is likely to remain the last release before Lean 4. We will use both Lean's basic libraries and `mathlib`.

Lean boasts the following strengths:

- It has a highly expressive logic based on the calculus of inductive constructions, a dependent type theory.
- It is extended with classical axioms and quotient types.
- 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.

Nonetheless, be aware that Lean is a research project, with some rough edges.

Lean's core library includes only basic algebraic definitions. More setup and especially tactics are found in Lean's mathematical library, `mathlib`.[4] It is more than a mathematical library; it provides a lot of basic automation on top of barebones Lean.

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

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

[4] `https://github.com/leanprover/mathlib/`

This Book This book is a companion to the course Logical Verification (LoVe) taught at the Vrije Universiteit Amsterdam. Despite our attempts to make it self-contained, unlikely to be suitable as a standalone Lean tutorial—we recommend *Theorem Proving in Lean* [1]—nor is it a substitute for attending class or doing the exercises. Ultimately, theorem proving can only be learned by doing.

The Lean files accompanying this book can be found in a public repository.[5] The files' naming scheme follows the book's chapters; thus, `love06_monads_demo.lean` is the main file associated with Chapter 6 ("Monads"), which is demonstrated in class; `love06_monads_exercise_sheet.lean` is the exercise sheet; and `love06_monads_homework_sheet.lean` is the homework sheet.

We have a huge debt to the authors of *Theorem Proving in Lean* [1] and *Concrete Semantics: With Isabelle/HOL* [17], who have taught us Lean and programming language semantics.

Special Symbols Lean lets us enter Unicode symbols such as $\mathbb{Z}$, $\mathbb{Q}$, and $\to$ using LaTeX-style macros. These symbols can be entered in Visual Studio Code by typing `\Z`, `\Q`, or `\->` 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 the book 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 can see the different ways in which it can be entered.

$\neg$	`\not`	$\wedge$	`\and`	$\vee$	`\or`	$\to$	`\->`
$\leftrightarrow$	`\<->`	$\forall$	`\fo`	$\exists$	`\ex`	$\leq$	`\<=`
$\geq$	`\>=`	$\neq$	`\neq`	λ	`\la`	Π	`\Pi`
$\leftarrow$	`\<-`	$\mapsto$	`\\|->`	$\Rightarrow$	`\=>`	$\Longrightarrow$	`\==>`
$\mathbb{N}$	`\N`	$\mathbb{Z}$	`\Z`	$\mathbb{Q}$	`\Q`	$\mathbb{R}$	`\R`
α	`\a`	β	`\b`	γ	`\g`	δ	`\de`
$_0$	`\0`	$_1$	`\1`	$_2$	`\2`	$_3$	`\3`

[5] `https://github.com/blanchette/logical_verification_2019`

Part I

Basics

Chapter 1

Definitions and Lemma Statements

We start by the basics of Lean, without carrying out actual proofs yet. We review how to define types and functions and how to state their intended properties as lemmas.

Lean's logical foundation is a rich formalism called the *calculus of inductive constructions*, whose defining feature is its support for *dependent types*. In this chapter, we restrict our attention to its simply (i.e., nondependently) typed fragment, which 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 programming languages (e.g., Python, C++11, Java 8). In a first approximation:

Lean = typed functional programming + 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 corresponds to the mathematical definition

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

If you are not familiar with typed functional programming, we recommend that you study a tutorial, such as the first six chapters of *Learn You a Haskell for Great Good!*[1]

1.1 Types and Terms

Simple type theory (also called *higher-order logic*) corresponds roughly to the simply typed λ-calculus [4] 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] `http://learnyouahaskell.com/`

Types *Types* are either 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 regarded as sets. We would normally define the types $\mathbb{Z}$, $\mathbb{Q}$, and `bool` in such a way 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; when studying them, it is important to keep them apart. Lean's types may be *interpreted* as mathematical sets, but they *are not* sets.

Higher-order types are types containing left-nested $\to$ arrows. Values of such types are functions that take other functions as arguments. Accordingly, the type $(\mathbb{Z} \to \mathbb{Z}) \to \mathbb{Z}$ 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 $\mathbb{Z}$.

Terms The *terms*, or expressions, of simple type theory consist of constants `c`, applications `t u`, λ-expressions λ`x, t`, and variables `x`, where `t` and `u` denote terms. We write `t : `σ to indicate that term `t` has type σ.

- A *constant* `c : `σ is a symbol of type σ whose meaning is fixed in the current logical context. For example, an arithmetic theory might be expected to contain constants such as `0 :` $\mathbb{Z}$, `1 :` $\mathbb{Z}$, `abs :` $\mathbb{Z} \to \mathbb{N}$, `square :` $\mathbb{N} \to \mathbb{N}$, and `prime :` $\mathbb{N} \to$ `bool`. Note that functions (e.g., `abs`, `square`) and predicates (e.g., `prime`) are considered constants.
- An *application* `t u`, where `t :` $\sigma \to \tau$ and `u :` σ, is a term of type τ denoting the result of applying the function `t` to the argument `u`—e.g., `prime 0`. No parentheses are necessary around the argument, unless it is a complex term—e.g., `prime (abs 0)`.
- A λ*-expression* λ`x :` σ`, t`, where `t :` τ, denotes the total function of type $\sigma \to \tau$ that maps each value `x` of type σ to `t`, where `t` is a term that may contain `x`. For example, λ`x :` $\mathbb{Z}$`, square (abs x)` denotes the function that maps (the value denoted by) `0` to (the value denoted by) `square (abs 0)`, and likewise `1` to `square (abs 1)`, and so on. A more mathematically oriented syntax would have been `x` $\mapsto$ `square (abs x)`, but this is not supported by Lean.
- A *variable* `x :` σ refers back to the input of a λ-expression λ`x :` σ`, t` enclosing it. In the expression λ`x :` $\mathbb{Z}$`, square (abs x)`, the second `x` is a variable that refers back to the λ binder's input `x`.

Applications and λ-expressions mirror each other: A λ-expression "builds" a function; an application "destructs" a function. Although our functions are unary (i.e., they take one argument), we can build *n*-ary functions by nesting λs, using a technique called *currying*. For example, λ`x :` σ`, (`λ`y :` τ`, x)` denotes the function of type $\sigma \to (\tau \to \sigma)$ that takes two arguments and that returns the first one. Strictly speaking, $\sigma \to (\tau \to \sigma)$ takes a single argument and returns a function, which in turn takes an argument. Applications work analogously: If `K` denotes the term λ`x :` $\mathbb{Z}$`, (`λ`y :` $\mathbb{Z}$`, x)`, then `K 1 = (`λ`y :` $\mathbb{Z}$`, 1)` and `(K 1) 0 = 1`. The function `K` in `K 1`, which is applied only to one argument, is said to be *partially applied*.

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

$$\begin{array}{rcl} \sigma \to \tau \to \upsilon & \text{for} & \sigma \to (\tau \to \upsilon) \\ \texttt{t u v} & \text{for} & \texttt{(t u) v} \\ \lambda \texttt{x} : \sigma, \lambda \texttt{y} : \tau, \texttt{t} & \text{for} & \lambda \texttt{x} : \sigma, (\lambda \texttt{y} : \tau, \texttt{t}) \end{array}$$

and also

$$\begin{array}{rcl} \lambda (\texttt{x} : \sigma)\ (\texttt{y} : \tau), \texttt{t} & \text{for} & \lambda \texttt{x} : \sigma, \lambda \texttt{y} : \tau, \texttt{t} \\ \lambda \texttt{x y} : \sigma, \texttt{t} & \text{for} & \lambda (\texttt{x} : \sigma)\ (\texttt{y} : \sigma), \texttt{t} \end{array}$$

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 `(+) x y`. The parentheses around the + sign are necessary to disable the parsing of + as an infix operator. Partial application is possible with this syntax. For example, `(+) 1` denotes the unary function that adds one to its argument. Other ways to write this function are λ`x, (+) 1 x` and λ`x, 1 + x`.

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

```
constants a b : ℤ
constant f : ℤ → ℤ
constant g : ℤ → ℤ → ℤ

#check λx : ℤ, g (f (g a x)) (g x b)
#check λx, g (f (g a x)) (g x b)
```

The first three lines introduce four constants (`a`, `b`, `f`, `g`), which can be used to form terms. The last two lines invoke the `#check` diagnosis command to type-check some terms and print their types.

The `constant`(`s`) command can be used even to declare types. The next example introduces a type of "trooleans" and three values of that type:

```
constant trool : Type
constants ttrue tfalse tmaybe : trool
```

The constants thus introduced have no definition. They are fully unspecified. For example, we cannot tell whether `ttrue`, `tfalse`, and `tmaybe` are equal or not, and we know nothing about `trool` except that it has at least one value (e.g., `ttrue`).

Type Checking and Type Inference When Lean parses a term, it checks whether the term is well typed. In the process, Lean will try to infer the types of bound variables if those are omitted—e.g., the type of `x` in λ`x, 1 + x`. Type inference helps keep notations lighter and saves some keystrokes.

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` : $\mathbb{N}$ and `0` : $\mathbb{R}$) can lead to undecidability [20]. Lean takes a pragmatic approach, by assuming that numerals `0`, `1`, `2`, etc., 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 $\Gamma \vdash \texttt{t} : \sigma$, meaning that term `t` has type σ in context Γ. The context gives the types of the variables in `t` that are not bound by any λ. There

are four typing rules, corresponding to the four kinds of terms:

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

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

$$\frac{\Gamma, \mathtt{x} : \sigma \vdash \mathtt{t} : \tau}{\Gamma \vdash (\lambda \mathtt{x} : \sigma, \mathtt{t}) : \sigma \to \tau}\ \textsc{Lam}$$

$$\frac{}{\Gamma \vdash \mathtt{x} : \sigma}\ \textsc{Var}\quad \text{if } \mathtt{x} : \sigma \text{ occurs in } \Gamma$$

The first and last rules, labeled Cst and Var, have no premises, but they have side conditions that must be satisfied for the rules to apply. The other two rules take one or two judgments as premises and produce a new judgment.

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, forming a *formal derivation* of a typing judgment. Like natural trees, derivation trees are drawn with their 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 $\lambda \mathtt{x} : \mathbb{Z}, \mathtt{abs\ x}$ has type $\mathbb{Z} \to \mathbb{N}$ in the empty variable context:

$$\dfrac{\dfrac{\dfrac{}{\mathtt{x} : \mathbb{Z} \vdash \mathtt{abs} : \mathbb{Z} \to \mathbb{N}}\ \textsc{Cst} \qquad \dfrac{}{\mathtt{x} : \mathbb{Z} \vdash \mathtt{x} : \mathbb{Z}}\ \textsc{Var}}{\mathtt{x} : \mathbb{Z} \vdash \mathtt{abs\ x} : \mathbb{N}}\ \textsc{App}}{\vdash (\lambda \mathtt{x} : \mathbb{Z}, \mathtt{abs\ x}) : \mathbb{Z} \to \mathbb{N}}\ \textsc{Lam}$$

Reading the proof from the root upwards, notice how Lam moves the variable bound by the λ-expression to the context, making an application of Var possible further up the tree.

Type inference is a generalization of type checking where the types on the right-hand side of the colon (:) in judgments may be replaced by placeholders. Lean's type inference is based on an algorithm due to J. Roger Hindley [11] and Robin Milner [15], which also forms the basis of Haskell, OCaml, and Standard ML. The algorithm generates type constraints involving placeholders α, β, γ, ..., and attempts to solve them using a type unification procedure. For example, when inferring the type α of $\lambda \mathtt{x}, \mathtt{abs\ x}$, Lean would perform the following schematic type derivation:

$$\dfrac{\dfrac{\dfrac{}{\mathtt{x} : \beta \vdash \mathtt{abs} : \beta \to \gamma}\ \textsc{Cst} \qquad \dfrac{}{\mathtt{x} : \beta \vdash \mathtt{x} : \beta}\ \textsc{Var}}{\mathtt{x} : \beta \vdash \mathtt{abs\ x} : \gamma}\ \textsc{App}}{\vdash (\lambda \mathtt{x}, \mathtt{abs\ x}) : \alpha}\ \textsc{Lam}$$

In addition, Lean would generate constraints to ensure that all the rule applications are legal:

1. For the LAM application, the type of λ`x, abs x` must have the form $\beta \to \gamma$, for some types β and γ. Thus, Lean would generate the constraint $\alpha = \beta \to \gamma$.

2. For the CST application, the type of `abs` must correspond to the declaration as $\mathbb{Z} \to \mathbb{N}$. Thus, Lean would generate the constraint $\beta \to \gamma = \mathbb{Z} \to \mathbb{N}$.

By putting the two constraints together, we obtain the solution $\alpha = \mathbb{Z} \to \mathbb{N}$, which is indeed the type that Lean infers for λ`x, abs x`.

Type Inhabitation Given a type σ, the type inhabitation problem consists of finding an "inhabitant" of that type—i.e., a term of type σ. It may seem like a pointless exercise, but as we will see in Chapter 3, there is a close connection between this problem and the problem of finding a proof of a proposition. In other words, seemingly silly exercises of the form "specify a term of type σ" are good practice towards mastery of theorem proving.

To create a term of a given type, you can 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 λ`x, _`, where `_` is a placeholder for a missing term. Lean will mark `_` as an error; if you hover over it in Visual Studio Code, a tooltip will appear, specifying the type of the missing term (here, τ) as well as any constants declared in the context.

2. Given a type σ (which may be a function type), you can use any constant `c` : $\tau_1 \to \cdots \to \tau_n \to \sigma$ to build a term of that type. For each of the n arguments to `c`, you need to pass a placeholder, yielding `c _ ... _`.

The placeholders can be eliminated recursively using the same procedure.

As an example, we will apply the procedure 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 possible, with σ := $\alpha \to \beta \to \gamma$ and τ := $((\beta \to \alpha) \to \beta) \to \alpha \to \gamma$. This results in the term λ`f, _`, which has the right type but has a placeholder left. Since the argument `f` has type $\alpha \to \beta \to \gamma$, a function type, it makes sense to use the name `f` for it. Then we continue recursively with the placeholder, of type $((\beta \to \alpha) \to \beta) \to \alpha \to \gamma$. Again, only step 1 is possible, so we end up with the term λ`f, `λ`g, _`, where the placeholder has type $\alpha \to \gamma$. A third application of step 1 yields the term λ`f, `λ`g, `λ`a, _`, where the placeholder has type γ.

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

$$\begin{aligned} &\texttt{f} : \alpha \to \beta \to \gamma \\ &\texttt{g} : (\beta \to \alpha) \to \beta \\ &\texttt{a} : \alpha \end{aligned}$$

Recall that we are trying to build a term of type γ. The only constant 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 λ`f, `λ`g, `λ`a, f _ _`.

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

We are now close to our goal. The remaining placeholder has type $\beta \to \alpha$, corresponding to `g`'s argument type. Applying step 1, we replace the placeholder with λ`b, _`, where the new placeholder has type α. Like above, we can supply `a`. Our final term is λ`f, λg, λa, f a (g (λb, a))`—i.e, λ`f g a, f a (g (λb, a))`.

The above derivation was tedious but deterministic: At each point, either step 1 or 2 was applicable, but never both. This will not always be the case. We might encounter deadends and need to backtrack. We may also fail altogether, with no possiblity to backtrack; for example, in the empty context, it is impossible to supply a witness for α.

1.2 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 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 equivalent names, including algebraic data types, inductive data types, freely generated data types, or simply "data types" or "datatypes."

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

```
inductive nat : Type
| 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)
```

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

The general format of all type declarations presented in this chapter is

```
inductive type-name : Type
| constructor-name₁ : type₁
    ⋮
| constructor-nameₙ : typeₙ
```

For the naturals, it is possible to convince Lean to use the familiar names ℕ, 0, 1, 2, etc., and indeed the built-in `nat` type offers such syntactic sugar. But using the lengthier notations sheds more light on the syntax of Lean's type definitions, and anyway syntactic sugar is known to cause cancer of the semicolon [19].

In the Lean file accompanying this chapter, the definition of `nat` is enclosed by a namespace, delimited by the commands `namespace my_nat` and `end my_nat`, to keep its effects contained to a portion of the file. After `end my_nat`, any occurrence of `nat`, `nat.zero`, or `nat.succ` refers to Lean's predefined type of natural numbers. Similarly, the entire file is enclosed 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 `my_nat` namespaces displays the following information:

```
inductive LoVe.my_nat.nat : Type
constructors:
LoVe.my_nat.nat.zero : nat
LoVe.my_nat.nat.succ : nat → nat
```

The focus on natural numbers is one of the many features of this book 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 13.

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. The next example shows how this could be done in Lean:

```
inductive aexp : Type
| num : ℤ → aexp
| var : string → aexp
| add : aexp → aexp → aexp
| sub : aexp → aexp → aexp
| mul : aexp → aexp → aexp
| 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`, we have that `num i` is an `aexp` value.
2. For every character string `x`, we have that `var x` is an `aexp` value.
3. If $\texttt{e}_1$ and $\texttt{e}_2$ are `aexp` values, then so are `add` $\texttt{e}_1$ $\texttt{e}_2$, `sub` $\texttt{e}_1$ $\texttt{e}_2$, `mul` $\texttt{e}_1$ $\texttt{e}_2$, and `div` $\texttt{e}_1$ $\texttt{e}_2$.

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

Comparison with Java At this point, it may be instructive to compare the concise Lean specification of `aexp` above with the canonical 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; }
    }
```

Admittedly, the Java version is more flexible, because users can add new "constructors" simply by providing new classes. But this extensionality comes at a price, especially when we want to reason logically about a program. Extensibility is possible in Lean as well if desired.

Comparison with C 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 {
    AET_NUM, AET_VAR, AET_ADD, AET_SUB, AET_MUL, AET_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, `num`:

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

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

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

Lists The next type we consider is that of finite lists:

```
inductive list (α : Type) : Type
| nil  : list
| cons : α → list → list
```

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. Inside the definition, because α is fixed on the left-hand side of the column, it is sufficient to write `list` to obtain `list` α.

The following commands allow us to inspect the constructors' types:

```
#check list.nil
#check list.cons
```

The output is

```
list.nil : Π(α : Type), list α
list.cons : ?M_1 → list ?M_1 → list ?M_1
```

Informally:

- The `nil` constructor takes a type α as argument and produces a result of type `list` α. (The Π syntax for the type of arguments will be explained in Chapter 3.)
- The `cons` constructor takes an element (the *head*) of some arbitrary type `?M_1` as argument and a list over `?M_1` (the *tail*) and produces a result of type `list ?M_1`. Unlike for `nil`, there is no need to pass a type argument to `cons`—the type is inferred from the first argument. If we want to pass the type argument explicitly, we need to write an at sign (`@`) in front of the constant: `@list.cons`. We then have the type `@list.cons` : Π`(`α `: Type),` $\alpha \to$ `list` $\alpha \to$ `list` α.

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

`[]`	for	`list.nil`
`h :: t`	for	`list.cons h t`
`[`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 :: reverse ys` is parsed as `(f x) :: (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 precedences; it is all too easy to misread `f x::reverse ys` as `f (x::reverse) ys`.

1.3 Function Definitions

If all we want is to declare a function, we can use the `constant` command described above (Section 1.1). But usually, we want to *define* the function's behavior, and for this we can use the `def` command.

Going back to the arithmetic expression example (Section 1.2), 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 e₁ + eval e₂
| (aexp.sub e₁ e₂) := eval e₁ - eval e₂
| (aexp.mul e₁ e₂) := eval e₁ * eval e₂
| (aexp.div e₁ e₂) := eval e₁ / eval e₂
```

The keyword `def` introduces the definition. The general format of pattern matching is

```
def name (vars₁ : type₁) ... (varsₘ : typeₘ) : type
| patterns₁ := result₁
    ⋮
| patternsₙ := resultₙ
```

Patterns may contain variables, whose scope is the corresponding right-hand side, as well as constructors. For example, in the second case of `eval`'s definition, the variable `x` can be used in the right-hand side `env x`.

Some definitions do not need pattern matching. For these, the syntax is simply

```
def name (vars₁ : type₁) ... (varsₘ : typeₘ) type :=
result
```

We can have pattern matching without recursion (e.g., in the `num` and `var` cases above), and we can have recursion without pattern matching.

Structural recursion is a form of recursion that allows us to peel off one or more constructors from the value on which we recurse. The `eval` function above is structurally recursive. Such functions are guaranteed to call themselves only finitely many times before the recursion stops. This is a prerequisite for establish that the function terminates, an important property to ensure logical consistency and termination of evaluation.

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:

```
-- illegal
def evil : ℕ → ℕ
| n := nat.succ (evil n)
```

If Lean were to accept this definition, we could exploit it to prove that `0 = 1`, by subtracting `evil n` on each side of the equation `evil n = nat.succ (evil n)`.

The basic arithmetic operations on natural numbers, such as addition, can be defined by structural recursion:

```
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 `succ` constructor from the second argument. (Given that addition is symmetric, there is no deep reason for choosing to recurse on the second argument.)

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

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

Both commands print `9`, as expected.

It is generally 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 `#reduce` or `#eval` 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)
```

This time, `#reduce` prints `14`:

```
#reduce mul 2 7
```

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

```
def power : ℕ → ℕ → ℕ
| _ 0            := 1
| m (nat.succ n) := m * power m n
```

Since the first argument does not change in the recursive calls, we can factor it out and put it next to the function's name, before the colon introducing the function's type:

```
def power₂ (m : ℕ) : ℕ → ℕ
| 0            := 1
| (nat.succ n) := m * power₂ n
```

In fact, we already saw a similar syntax for the type argument α of the `list` constructor (Section 1.2). 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
| 0            := z
| (nat.succ n) := f (iter n)
```

```
def power₃ (m n : ℕ) : ℕ :=
iter ℕ 1 (λl, m * l) n
```

Notice that the second definition is not recursive.

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 xs ys)

#check append
#reduce append _ [3, 1] [4, 1, 5]
```

The `append` function takes three arguments: a type α and two lists of type `list` α. By passing the placeholder `_`, we leave it to Lean to infer the type α from the type of the other two arguments.

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

```
def append₂ {α : Type} : list α → list α → list α
| list.nil         ys := ys
| (list.cons x xs) ys := list.cons x (append₂ xs ys)

#check append₂
#reduce append₂ [3, 1] [4, 1, 5]

#check @append₂
#reduce @append₂ _ [3, 1] [4, 1, 5]
```

The at sign (`@`) can be used to make the implicit arguments explicit.

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 append₃ {α : Type} : list α → list α → list α
| []        ys := ys
| (x :: xs) ys := x :: append₃ xs ys
```

In Lean's standard library, `++` is provided as syntactic sugar for the append function. 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]
```

1.4 Lemma Statements

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

In Lean, propositions are simply terms of type `Prop`. A proposition that can be proved is a theorem (or lemma, corollary, etc.); otherwise it is a nontheorem

or false statement. Mathematicians sometimes use "proposition" as a synonym for theorem (e.g., "Proposition 3.14"), but in formal logic it is customary to let it encompass false statements.

Here are examples of true statements that can be made about the addition, multiplication, and list reversal operations defined in Section 1.3:

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

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


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

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

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


lemma reverse_reverse {α : Type} (xs : list α) :
  reverse (reverse xs) = xs :=
sorry
```

The general format is

```
lemma name (vars₁ : type₁) ... (varsₘ : typeₘ) statement := proof
```

The `:=` symbol separates the lemma's statement and its proof. 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 an actual proof. In Chapters 2 and 3, we will see how to eliminate these `sorry`s.

1.5 Summary of New Syntax

At the end of this and other chapters, we include a brief summary of the syntax introduced in the chapter. Some commands have multiple meanings, which will be introduced gradually. We refer to *The Lean Reference Manual* [3] and the *Theorem Proving in Lean* [1] and *Programming in Lean* [2] tutorials for details and to Lean's source code for the ultimate truth.

Declarations

`constant(s)`	declares unspecified new constants or types
`def`	defines a new constant
`inductive`	introduces a type and its constructors
`lemma`	states a lemma
`namespace ... end`	puts all declarations in a named scope (to avoid clashes)

Proof Commands

`sorry`	stands for a missing proof or definition

Diagnosis Commands

`#check`	checks and prints type of a term
`#eval`	executes a term using an optimized interpreter
`#print`	prints definition of a constant or function
`#reduce`	executes a term using Lean's kernel

Chapter 2

Tactical Proofs

In this chapter, we see how to prove Lean lemmas using tactics, and we review the most important Lean tactics. A *tactic* is a procedure that operates on the goal—the formula to prove—and either fully solves it or produces new subgoals. When we state a lemma, the lemma statement is the initial goal. A proof is complete once all (sub)*goals have been eliminated using suitable tactics.

The tactics described here are documented in more detail in Chapter 5 of *Theorem Proving in Lean* [1] and Chapter 6 of *The Lean Reference Manual* [3].

2.1 Tactic Mode

In Chapter 1, whenever a proof was needed, we simply put a `sorry` placeholder. For a tactical proof, we will now write `begin` and `end` instead to enter and leave *tactic mode*. In tactic mode, we can apply a sequence of tactics, separated by commas (`,`).

Tactics operate on the proof goal, which consists of the conjecture χ that we want to prove and of a context Γ that provides constant declarations of the form `c` : τ and hypotheses of the form `h` : φ. We write `C`, `H` $\vdash \chi$ to denote a goal, where `C` is a set of (local) constants, `H` is a set of (local) hypotheses, and χ is the goal's conclusion. The constants and hypotheses form a set; the comma is read as set union. We call `C`, `H` the *local context* of the goal.

This will become clearer with an example:

```
lemma fst_of_two_props :
  ∀a b : Prop, a → b → a :=
begin
  intros a b,
  intros ha hb,
  apply ha
end
```

Note that the implication arrow $\rightarrow$ is right-associative; this means that `a` $\rightarrow$ `b` $\rightarrow$ `a` is the same as `a` $\rightarrow$ `(b` $\rightarrow$ `a)`. In the example, three tactics are invoked, each on its own line. Let us trace their behavior:

1. Initially, the goal is simply the lemma statement:

   $$\vdash \forall \texttt{a b : Prop, a} \rightarrow \texttt{b} \rightarrow \texttt{a}$$

2. The `intros a b` tactic tells Lean to fix two arbitrary constants, `a` and `b`, corresponding to the two bound variables of the same names. It is customary, although not mandatory, to name the constants after the bound variables. The tactic mimics how

mathematicians work on paper: To prove a ∀-quantified formula, it suffices to prove it for some arbitrary value of the bound variable. The goal becomes

```
a b : Prop ⊢ a → b → a
```

3. Next, the `intros ha hb` tactic tells Lean to move the assumptions `a` and `b` to the context and to call these hypotheses `ha` and `hb`. Indeed, to prove an implication, we can take its left-hand side as hypothesis and try to prove its right-hand side. The goal becomes

```
(a b : Prop) (ha : a) (hb : b) ⊢ a
```

4. The `apply ha` tactic tells Lean to match `ha : a` against the goal `a`. Since `a` is syntactically equal to `a`, this finishes the proof.

We can save ourselves the `intros` invocations by declaring the constants and hypotheses as parameters of the lemma, as follows:

```
lemma fst_of_two_props₂ (a b : Prop) (ha : a) (hb : b) :
  a :=
begin
  apply ha
end
```

2.2 Basic Tactics

We already saw the `intros` and `apply` tactics. These are the staples of tactical proofs. Other basic tactics include `exact`, `assumption`, `refl`, `ac_refl`, and `use`.

These basic 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.

intro(s)

```
intro [name]
intros [name₁ ... nameₙ]
```

(The symbols [ ] encompass optional syntax, here and elsewhere.) The `intro` tactic moves the leading ∀-quantified variable or the leading assumption `a →` from the conclusion 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. The `intros` variant can be used to move several variables or assumptions at once.

apply

```
apply lemma
```

The `apply` tactic matches the goal's conclusion with the conclusion of the specified lemma and adds the lemma's hypotheses as new goals.

exact

```
exact lemma
```

The `exact` tactic matches the goal's conclusion with the specified lemma, closing the goal. We can often use `apply` in such situations, but `exact` communicates our intentions better. In the example from Section 2.1, we could have used `exact ha` instead of `apply ha`.

assumption

The `assumption` tactic finds a hypothesis from the local context that matches the goal's conclusion and applies it to solve the goal. In the example from Section 2.1, we could have used `assumption` instead of `apply ha`.

refl

The `refl` tactic solves goals of the `l = r`, where the two sides are equal *up to computation*. We also say, equivalently, that the two terms are *definitionally equal*. Computation means unfolding of definitions, reduction of an application of a λ-expression to an argument, and more. These conversions have traditional names. The main conversions are listed below together with examples, in a context with `def double (n : ℕ) : ℕ := n + n`:

α-conversion	`(λx, f x) = (λy, f y)`
β-conversion	`(λx, f x) a = f a`
δ-conversion	`double m = m + m`
ζ-conversion	`(let n : ℕ := 2 in n + n) = 4`
η-conversion	`(λx, f x) = f`
ι-conversion	`prod.fst (a, b) = a`

ac_refl

Proves `l = r`, where the two sides are equal up to associativity and commutativity. This works for binary operations that are registered as associative and commutative (e.g., `+` and `*` on arithmetic types). We will see an example in Section 2.6.

use

```
use term
```

The `use` tactic allows us to supply a witness for an existential quantifier. In the example below, where `def double (n : ℕ) : ℕ := n + n`, the witness is `0`, and the proof goes through by computation once the witness has been supplied:

```
lemma nat_exists_double_iden :
  ∃n : ℕ, double n = n :=
begin
  use 0,
  refl
end
```

2.3 Proofs about Logical 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 ($\wedge$).

```
lemma and_swap (a b : Prop) :
  a ∧ b → b ∧ a :=
begin
  intro hab,
  apply and.intro,
  apply and.elim_right,
  exact hab,
  apply and.elim_left,
```

```
    exact hab
  end
```

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 immediately after each comma, you can see the effect of the command on that line.

The proof is a typical combination `intro`, `apply`, and `exact` tactics. The lemmas used are

```
and.intro          : ?a → ?b → ?a ∧ ?b
and.elim_left      : ?a ∧ ?b → ?a
and.elim_right     : ?a ∧ ?b → ?b
```

where the question marks indicate variables that can be instantiated—for example, by matching the goal's conclusion against one the conclusion of a lemma.

The three lemmas above are the introduction rule and the two elimination rules associated with conjunction. An *introduction rule* for a symbol (e.g., $\wedge$) is a lemma whose conclusion contains that symbol. Dually, an *elimination rule* has the symbol in an assumption. In the above proof, we apply the introduction rule associated with $\wedge$ to prove the goal `b ∧ a`, and we apply the two elimination rules to extract `b` and `a` from the hypothesis `a ∧ b`.

The following is an alternative proof of the same lemma:

```
lemma and_swap₂ :
  ∀a b : Prop, a ∧ b → b ∧ a :=
begin
  intros a b hab,
  apply and.intro,
  { exact and.elim_right hab },
  { exact and.elim_left hab }
end
```

The lemma is stated differently, with `a` and `b` as $\forall$-quantified variables instead of parameters of the lemma. Logically, this is equivalent, but in the proof we must then introduce (using `intro`) `a` and `b` in addition to `a ∧ b`.

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 solve it. In our example, the `apply and.intro` invocation creates two subgoals, `b` and `a`.

The third difference between the proofs is that we now apply `and.elim_right` and `and.elim_left` directly to the hypothesis `a ∧ b` to obtain `b` and `a`, respectively, instead of the combination of `apply` and `exact` used in the first proof. Quantifier instantiation has the same syntax—juxtaposition—as passing an argument to a function. This is not a coincidence, as we will see in Chapter 3.

The introduction and elimination rules for disjunction ($\vee$) are as follows:

```
or.intro_left      : ∀b : Prop, ?a → ?a ∨ b
or.intro_right     : ∀b : Prop, ?a → b ∨ ?a
or.elim            : ?a ∨ ?b → (?a → ?c) → (?b → ?c) → ?c
```

The `or.elim` rule may seem counterintuitive at a 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.

The introduction and elimination rules for equivalence ($\leftrightarrow$) are as follows:

```
iff.intro          : (?a → ?b) → (?b → ?a) → (?a ↔ ?b)
iff.elim_left      : (?a ↔ ?b) → ?a → ?b
iff.elim_right     : (?a ↔ ?b) → ?b → ?a
```

The introduction and elimination rules for existential quantification (∃) are as follows:

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

The introduction rule for ∃ is what the `use` tactic uses internally. For example:

```
lemma nat_exists_double_iden₂ :
  ∃n : ℕ, double n = n :=
begin
  apply exists.intro 0,
  refl
end
```

The elimination rule for ∃ has a similar "counterintuitive" flavor to the elimination rule for ∨.

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

```
true.intro        : true
```

For falsehood (`false`), there is only an elimination rule:

```
false.elim        : false → ?a
```

Negation (`not`) is defined in terms of implication and falsehood: ¬ φ abbreviates φ → `false`. Lean's logic is classical, with support for the law of excluded middle and proofs by contradiction:

```
classical.em      : ∀a : Prop, a ∨ ¬ a
classical.by_contradiction
                  : (¬ ?a → false) → ?a
```

Finally, implication (→) and universal quantification (∀) are the dogs that did not bark. For both of them, `intro` is the introduction rule, and application (juxtaposition) is the elimination rule. For example, if we have `hab : a → b` and `ha : a`, then `hab ha : b` is a proof of `b`.

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

- If the goal's conclusion is an implication φ → ψ, invoke `intro hφ` to move φ into your hypotheses: `...` `(hφ : φ)` ⊢ ψ.
- If the goal's conclusion is a universal quantification ∀`x` : σ, φ, invoke `intro x` to move it into the local context: `...` `(x :` σ`)` ⊢ φ.
- Otherwise, look for a hypothesis or a lemma whose conclusion has the same shape as the goal's conclusion (possibly containing variables that can be matched against the goal), and `apply` it. For example, if the goal's conclusion is ⊢ ψ and you have a hypothesis `hφψ` : φ → ψ, try `apply hφψ`.
- A negated goal ⊢ ¬ φ is definitionally equal to ⊢ φ → `false`, so you can invoke `intro hφ` to produce the subgoal `hφ` : φ ⊢ `false`. Expanding negation's definition by invoking `dunfold not` (described in Section 2.4) is often a good strategy.
- Sometimes you can make progress by replacing the goal by `false`, by entering `apply false.elim`. As next step, you would typically `apply` a hypothesis of the form φ → `false` or ¬ φ.
- When you face several choices (e.g., between `or.intro_left` and `or.intro_right`), remember which choices you have made, and backtrack when you reach a dead end or have the impression you are not making any progress.

- If you have difficulties carrying out a proof, it can be a good idea to check whether the goal actually is provable under the given assumptions. Be also aware that even if you started with a provable lemma statement, it is possible that the goal is not provable (e.g., if you used "unsafe" rules such as `or.intro_left`).

2.4 Rewriting Tactics

The rewriting tactics replace equals for equals. By default, they apply to the goal's conclusion, but they can also be applied to hypotheses specified using an `at` clause:

`at` ⊢	applies to the conclusion
`at` $\mathtt{h}_1 \ldots \mathtt{h}_n$	applies to the specified hypotheses
`at *`	applies to all possible hypotheses and to the conclusion

rw

`rw` *lemma* [`at` *position*]

The `rw` tactic applies a single equation as a left-to-right rewrite rule, once. To apply a lemma as a right-to-left rewrite rule, put a short left arrow (←) in front of its name, or pass the lemma to `eq.symm`.

For example, given the lemma `l : b = a` and the goal ⊢ `h a b b`, the tactic invocation `rw l` produces the subgoal ⊢ `h a a b`, whereas `rw ←l` or `rw (eq.symm l)` produces the subgoal ⊢ `h b b b`.

`rw [`$lemma_1$`, ..., `$lemma_n$`]` [`at` *position*]

The above syntax abbreviates `rw` $lemma_1$`, ...,` `rw` $lemma_n$. (The thick, small brackets `[]` are part of the syntax.)

simp

`simp` [`at` *position*]

The `simp` tactic applies a standard set of rewrite rules, called the *simp set*, exhaustively. The simp set can be extended by putting the `@[simp]` attribute on lemmas. The `simp` tactic is more powerful than `rw` because it can rewrite terms containing bound variables (e.g., bound by λ, ∀, or ∃).

`simp [`$lemma_1$`, ..., `$lemma_n$`]` [`at` *position*]

For the above `simp` variant, the specified lemmas are temporarily added to the simp set. In the lemma list, an asterisk (*) can be use to represent all local hypotheses. The minus sign (-) in front of a lemma name temporarily removes the lemma from the simp set. To apply a lemma as a right-to-left rewrite rule, prefix its name with a short left arrow (←).

For example, given the lemma `l : b = a` and the goal ⊢ `h a b b`, the tactic invocation `simp [l]` produces the subgoal ⊢ `h a a a`, where both occurrences of `b` have been replaced by `a`, whereas `simp [←l]` produces the subgoal ⊢ `h b b b`.

To find out exactly what `simp` does, you can enable tracing by adding the command `set_option trace.simplify.rewrite true` to your Lean file.

dunfold

`dunfold` $constant_1 \ldots constant_n$ [`at` *position*]

The `dunfold` tactic expands the definition of one or more constants that are specified without pattern matching (e.g., `not`).

2.5 Induction Tactic

induction

```
induction variable
```

The `induction` tactic performs structural induction on the specified variable. This gives rise to as many subgoals as there are constructors in the definition of the variable's type. Induction hypotheses are available as hypotheses in the subgoals corresponding to recursive constructors (e.g., `nat.succ` or `list.cons`).

2.6 Proofs about Natural Numbers

Equipped with a tactic for structural induction, we can now carry out meaningful proofs about the addition and multiplication operations we defined by structural recursion in Section 1.3.

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

```
lemma add_zero (n : ℕ) :
  add 0 n = n :=
begin
  induction n,
  { refl },
  { simp [add, n_ih] }
end
```

The name `n_ih` is generated by the `induction` tactic. It refers to the induction hypothesis for the induction on `n`. We can supply our own names, and address the cases in an arbitrary order, by using the `case` keyword in front of each case, together with the name of the case and the desired names for the local constants and hypotheses introduced by the tactic. For example:

```
lemma add_zero₂ (n : ℕ) :
  add 0 n = n :=
begin
  induction n,
  case nat.zero {
    refl },
  case nat.succ : m ih {
    simp [add, ih] }
end
```

Instead of `n` and `n_ih`, we choose the names `m` and `ih`.

We can write the proof more concisely if we want:

```
lemma add_zero₃ (n : ℕ) :
  add 0 n = n :=
by induction n; simp [add, *]
```

The syntax `by` *tactic* abbreviates `begin` *tactic* `end`. It can be used when a single (possibly complex) tactic suffices to solve the goal.

The semicolon (`;`) is a tactic combinator. The complex tactic $tactic_1$; $tactic_2$ first applies $tactic_1$ and then applies $tactic_2$ to all subgoals emerging from $tactic_1$. Our example works because `simp [add, *]` solves not only the induction step but also the base case (for which `refl` would have sufficed).

We can keep on proving lemmas by structural induction:

```
lemma add_succ (m n : ℕ) :
  add (nat.succ m) n = nat.succ (add m n) :=
begin
  induction n,
  case nat.zero {
    refl },
  case nat.succ : m ih {
    simp [add, ih] }
end

lemma add_comm (m n : ℕ) :
  add m n = add n m :=
begin
  induction n,
  case nat.zero {
    simp [add, add_zero] },
  case nat.succ : m ih {
    simp [add, add_succ, ih] }
end

lemma add_assoc (l m n : ℕ) :
  add (add l m) n = add l (add m n) :=
begin
  induction n,
  case nat.zero {
    refl },
  case nat.succ : m ih {
    simp [add, ih] }
end
```

Once we have proved that a binary operator is commutative and associative, it is a good idea to let Lean's automation (notably `ac_refl`) know about this. Here is the syntax to achieve this for `add`:

```
instance : is_commutative ℕ add := { comm := add_comm }
instance : is_associative ℕ add := { assoc := add_assoc }
```

Here is an example that uses `ac_refl` to reason up to associativity and commutatitivy of `add`:

```
lemma mul_add (l m n : ℕ) :
  mul l (add m n) = add (mul l m) (mul l n) :=
begin
  induction n,
  case nat.zero {
    refl },
  case nat.succ : m ih {
    simp [add, mul, ih],
    ac_refl }
end
```

2.7 Goal Management Tactics

The following tactics help manage the goal, by allowing us to give more meaningful names to local constants or hypotheses, remove useless hypotheses, or undo the effect of an `intro` invocation.

rename

```
rename constant-or-hypothesis new-name
```

The `rename` tactic renames a local constant or hypothesis.

clear

```
clear constant-or-hypothesis₁ ... constant-or-hypothesisₙ
```

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

revert

```
revert constant-or-hypothesis₁ ... constant-or-hypothesisₙ
```

The `revert` tactic performs the opposite of `intros`: Moves the specified local constants and hypotheses into the goal's conclusion using universal quantification (∀) for constants and implication (→) for hypotheses.

For example, if the goal is `a : Prop, ha : a ⊢ b`, the tactic invocation `revert a ha` produces the subgoal `⊢ ∀a : Prop, a → b`.

2.8 Summary of New Syntax

Commands

`set_option`	changes or activates tracing, syntax output, etc.

Proof Commands

`begin ... end`	applies a list of tactics in sequence
`by`	applies a single tactic

Tactic Combinators

`;`	applies the second tactic to all subgoals yielded by the first tactic
`{ ... }`	focuses on the first subgoal; needs to solve that goal

Tactics

`ac_refl`	proves `l = r` if `l` equals `r` up to associativity and commutativity
`apply`	matches the goal's conclusion with the lemma's conclusion
`assumption`	solves the goal using a local hypothesis
`clear`	removes a local constant or hypothesis from the goal
`dunfold`	unfolds a definition without pattern matching (e.g., `not`)
`exact`	solves the goal by the specified term
`induction`	performs structural induction on a variable of an inductive type
`intro(s)`	moves ∀-quantified variables into the goal's hypotheses
`refl`	proves `l = r` where `l` and `r` are equal up to computation
`rename`	renames a local constant or hypothesis
`revert`	undoes the effect of `intro`
`rw`	applies the given rewrite rules once, in sequence
`simp`	applies a set of preregistered rewrite rules exhaustively
`use`	provides a witness to an existential quantifier

Chapter 3

Structured Proofs and Proof Terms

Tactics are a *backward* (or *bottom-up*) proof mechanism. They start from the goal and break it down. Often it makes sense to work *forward*: to start with what we already know and proceed step by step towards our goal. *Structured proofs* are a style that supports this reasoning. They can be combined with the tactical style. Backward proofs tend to be easier to write but harder to read. It is, after all, easier to destruct things than to construct them.

Structured proofs are syntactic sugar sprinkled over Lean's *proof terms*. All proofs, whether tactical or structured, are reduced internally to proof terms. As the name suggests, these proofs are represented as terms. We have seen some proof terms already, in Chapter 2: If we have `hab : a → b` and `ha : a`, then `hab ha` is a proof term for `b`, and we write `hab ha : b`.

The concepts covered here are described in more detail in Chapters 2 to 4 of *Theorem Proving in Lean* [1]. Nederpelt and Geuvers's textbook [16] and Van Raamdonk's lecture notes [22] are other useful references.

3.1 Structured Proofs

Structured proofs are built using a number of constructs. We review the main ones below.

Lemma Name

The simplest kind of structured proof is the name of a lemma. If we have

```
lemma some_name : some_proposition :=
...
```

then `some_name` can be used as a proof of `some_proposition` later—for example:

```
lemma some_other_name : some_proposition :=
some_name
```

We can do the same with local hypotheses, and we can pass arguments to lemmas or hypotheses to instantiate ∀ quantifiers and discharge assumptions of a lemma.

As an more realistic example, suppose we have a lemma `add_comm (m n : ℕ) : add m n = add n m`, and suppose we want to prove its instance `add 0 n = add n 0`. This can be achieved neatly using the name of the lemma and two arguments:

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

This has the same effect as the tactical proof `by exact add_comm 0 n`, but is more concise.

assume

```
assume name₁ ... nameₙ,
```

The `assume` command moves variables or assumptions from the goal into the local context, as local constants or hypotheses. It can be seen as a structured version of the `intros` tactic. At the proof term level, it amounts to n λ binders.

have

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

The `have` command lets us state and prove an intermediate lemma statement, which can refer to the local context. Often we put a `_` placeholder for *proposition* if it can be inferred from the proof.

let

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

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 for computable data, not a proof. Often we omit the type if it can be inferred from the term. Folding or unfolding a `let` corresponds to ζ-conversion (Section 2.2).

show

```
show proposition, from structured-proof
show proposition, begin tactics end
show proposition, by tactic
```

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 an equal form up to computation. Instead of the syntax `show` *proposition*`, from` *structured-proof*, we can simply write *structured-proof* if we do not need to rephrase the goal.

The second and third syntaxes listed above remind us that it is possible to mix proof styles. Similarly, the *proof* part of a `have` command can be either a structured proof or a tactical proof introduced by `begin` or `by`. Generally, we tend to use structured proofs to sketch the main argument and resort to tactical proofs for proving subgoals or straightforward lemmas.

Another kind of mixture arises when we pass arguments to lemma names. For example, given, `hab : a → b` and `ha : a`, the tactic invocation `exact hab ha` will prove the goal `b`: `hab ha` is a (admittedly small) proof term nested in a tactic. Yet another kind of mixture arises when we open a `begin ... end` block after invoking some structured proof commands such as `assume` and `have`, as in the following example:

```
lemma and_swap₂ (a b : Prop) :
  a ∧ b → b ∧ a :=
assume hab : a ∧ b,
have ha : a := and.elim_left hab,
have hb : b := and.elim_right hab,
begin
  apply and.intro,
  { exact hb },
  { exact ha }
end
```

Here is a purer proof:

```
lemma and_swap (a b : Prop) :
  a ∧ b → b ∧ a :=
assume hab : a ∧ b,
have ha : a := and.elim_left hab,
have hb : b := and.elim_right hab,
show b ∧ a, from and.intro hb ha
```

Forward Tactics

Two of the proof commands presented above—`have` and `let`—are also available as tactics. Many users, especially beginners, prefer the tactical mode. Even with tactics, it can be useful to reason in a forward fashion. Observe that the syntax for the tactic `let` is slightly different than for the structured proof command of the same name, with a comma (`,`) instead of the keyword `in`.

`have` $\big[\big[$*name* `:`$\big]$ *proposition*$\big]$ `:=` *proof* `,`

`let` *name* $\big[$`:` *type*$\big]$ `:=` *term* `,`

3.2 Calculational Proofs

In informal mathematics, we often use transitive chains of equalities, inequalities, or equivalences (e.g., `a = b = c`, $\texttt{a} \geq \texttt{b} \geq \texttt{c}$, or $\texttt{a} \leftrightarrow \texttt{b} \leftrightarrow \texttt{c}$). In Lean, such *calculational proofs* are supported by the `calc` command. It provides a lightweight syntax and takes care of applying transitivity lemmas for preregistered relations (including equality and the arithmetic comparison operators).

The general syntax is as follows:

`calc` $expr_0\ op_1\ expr_1 : proof_1$
$\ldots\ op_2\ expr_2 : proof_2$
$\vdots$
$\ldots\ op_n\ expr_n : proof_n$

The horizontal dots (`...`) are part of the syntax. Each $proof_i$ establishes the statement $expr_{i-1}\ op_i\ expr_i$. The operators op_i need not be identical, but they need to be compatible. For example, $=$, $<$, and $\leq$ are compatible with each other, whereas $>$ and $<$ are not.

A simple example follows:

```
lemma two_mul_example (m n : ℕ) :
  2 * m + n = m + n + m :=
calc 2 * m + n = (m + m) + n :
  by rw two_mul
... = m + n + m :
  by ac_refl
```

The horizontal dots stand for the term `(m + m) + n`, which we would have had to repeat had we performed the proof with two `have`s:

```
lemma two_mul_example₂ (m n : ℕ) :
  2 * m + n = m + n + m :=
have h₁ : 2 * m + n = (m + m) + n := by rw two_mul,
have h₂ : (m + m) + n = m + n + m := by ac_refl,
show _, from eq.trans h₁ h₂
```

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

3.3 Induction by Pattern Matching

In Section 2.5, we saw how to use the `induction` tactic to carry out proofs by induction. An alternative, more structured style relies on pattern matching, as in the example below:

```
lemma add_zero :
  ∀n : ℕ, add 0 n = n
| 0            := by refl
| (nat.succ m) := by simp [add, add_zero m]
```

The patterns on the left, `0` and `(nat.succ m)`, correspond to the two constructors of the type of the ∀-quantified variable `n : ℕ`. On the right-hand side of each `:=` symbol is a proof for the corresponding case: the base case or the induction step.

Inside the induction step's proof, the induction hypothesis is available under the same name as the lemma we are proving. We explicitly pass `m` as argument to the hypothesis: This is useful as documentation, and it also prevents Lean from accidentally invoking the induction hypothesis in a circular or otherwise ill-founded fashion—for example, on `nat.succ m`. Lean's termination checker would detect this and raise an error.

Simultaneous pattern matching is also possible, as in the examples below:

```
lemma add_succ :
  ∀m n : ℕ, add (nat.succ m) n = nat.succ (add m n)
| m 0            := by refl
| m (nat.succ n) := by simp [add, add_succ m n]

lemma add_comm :
  ∀m n : ℕ, add m n = add n m
| m 0            := by simp [add, add_zero]
| m (nat.succ n) := by simp [add, add_succ, add_comm m n]

lemma add_comm₂ :
  ∀m n : ℕ, add m n = add n m
| m 0            := by simp [add, add_zero]
| m (nat.succ n) :=
  have ih : _ := add_comm₂ m n,
  by simp [add, add_succ, ih]

lemma add_assoc :
  ∀l m n : ℕ, add (add l m) n = add l (add m n)
| l m 0            := by refl
| l m (nat.succ n) := by simp [add, add_assoc l m n]
```

Notice the structured proof command `have` in the proof of `add_comm₂`.

The general format of lemmas with pattern matching is as follows:

```
lemma name (vars₁ : type₁) ... (varsₘ : typeₘ) : statement
| patterns₁ := proof₁
    ⋮
| patternsₙ := proofₙ
```

A few hints on how to carry out proofs by induction follow:

- It is usually beneficial to perform induction following the structure of the definition of one of the functions appearing in the goal.
- If the base case of an induction is difficult, this is often a sign that the wrong variable was chosen or that some lemmas are missing.

3.4 Dependent Types

Dependent types are the defining feature of the *dependent type theory* family of logics. Consider a function `pick` that takes a natural number `x` (a value from $\mathbb{N}$ = {0, 1, 2, ...}) and that returns a natural number between 0 and `x`. Conceptually, `pick` has a dependent type, namely, $(\mathtt{x} : \mathbb{N}) \to \{\mathtt{y} : \mathbb{N} \mathbin{//} \mathtt{y} \leq \mathtt{x}\}$. Informally, we read $\{\mathtt{y} : \mathbb{N} \mathbin{//} \mathtt{y} \leq \mathtt{x}\}$ as "the type consisting of all `y` from $\mathbb{N}$ such that $\mathtt{y} \leq \mathtt{x}$." We can think of this type as a $\mathbb{N}$-indexed family, where each member's type may depend on the index: $\mathtt{pick}\ \mathtt{x} : \{\mathtt{y} : \mathbb{N} \mathbin{//} \mathtt{y} \leq \mathtt{x}\}$. The name `x` is immaterial; we could have used any other name except `y`.

Unless otherwise specified, a *dependent type* means a type depending on a term, as in the above example, with $\mathtt{x} : \mathbb{N}$ as the term and $\{\mathtt{y} : \mathbb{N} \mathbin{//} \mathtt{y} \leq \mathtt{x}\}$ as the type that depends on it. This is what we mean when we claim that simple type theory does not support dependent types. But a type may also depend on another type—for example, the type constructor `list`, its η-expanded variant $\lambda\alpha : \mathtt{Type},\ \mathtt{list}\ \alpha$, or the polymorphic type $\lambda \alpha : \mathtt{Type},\ \alpha \to \alpha$ of functions with the same domain and codomain. A term may depend on a type—for example, the polymorphic identity function $\lambda\alpha : \mathtt{Type},\ \lambda\mathtt{x} : \alpha,\ \mathtt{x}$. And of course, a term may also depend on a term—for example, $\lambda\mathtt{x} : \mathbb{N},\ \mathtt{x}$.

In summary, there are four cases for $\lambda\mathtt{x},\ \mathtt{t}$ in dependent type theory:

Body (`t`)		Argument (`x`)	Description
A term	depending on	a term	Simply typed λ-abstraction
A type	depending on	a term	Dependent type (stricto senso)
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 Lam rules presented in Section 1.1 must be generalized to work with dependent types:

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

$$\frac{\Gamma, \mathtt{x} : \sigma \vdash \mathtt{t} : \tau[\mathtt{x}]}{\Gamma \vdash (\lambda\mathtt{x} : \sigma, \mathtt{t}) : (\mathtt{x} : \sigma) \to \tau[\mathtt{x}]}\ \textsc{Lam}'$$

The simply typed case arises when `x` does not occur in $\tau[\mathtt{x}]$. Then, we can simply write $\sigma \to$ and not $(\mathtt{x} : \sigma) \to$. The notation $\sigma \to \tau$ is equivalent to $(\_ : \sigma) \to \tau$. It is easy to see that App′ and Lam′ are identical to App and Lam when `x` does not occur in $\tau[\mathtt{x}]$.

The example below demonstrates App′:

$$\frac{\vdash \mathtt{pick} : (\mathtt{x} : \mathbb{N}) \to \{\mathtt{y} : \mathbb{N} \mathbin{//} \mathtt{y} \leq \mathtt{x}\} \qquad \vdash 5 : \mathbb{N}}{\vdash \mathtt{pick}\ 5 : \{\mathtt{y} : \mathbb{N} \mathbin{//} \mathtt{y} \leq 5\}}\ \textsc{App}'$$

The next example demonstrates Lam′:

$$\frac{\dfrac{\alpha : \mathtt{Type},\ \mathtt{x} : \alpha \vdash \mathtt{x} : \alpha}{\alpha : \mathtt{Type} \vdash (\lambda\mathtt{x} : \alpha,\ \mathtt{x}) : \alpha \to \alpha}\ \textsc{Lam or Lam}'}{(\lambda\alpha : \mathtt{Type},\ \lambda\mathtt{x} : \alpha,\ \mathtt{x}) : (\alpha : \mathtt{Type}) \to \alpha \to \alpha}\ \textsc{Lam}'$$

We conclude with an important remark on notations. Regrettably, the intuitive syntax $(\mathtt{x} : \sigma) \to \tau$ is not available in Lean. Instead, we must write $\Pi\mathtt{x} : \sigma,\ \tau$ to specify a dependent type. The arrow notation $\sigma \to \tau$ (without the `x`) is syntactic sugar for $\Pi\_ : \sigma,\ \tau$. The Π binder emphasizes that `x` is a bound variable within Π's scope.

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

As an alias for Π, we can also write $\forall$. The two symbols are fully interchangeable.

In Section 1.2, we saw the commands

```
#check list.nil
#check list.cons
```

and their output

```
list.nil : Π(α : Type), list α
list.cons : ?M_1 → list ?M_1 → list ?M_1
```

We can now make sense of the Π-type. Namely, `list.nil` is a function that takes a type α as argument and that returns a value of type `list` α.

3.5 The Curry–Howard Correspondence

You likely have noticed that we use the same symbol $\to$ both as the implication symbol and as the type constructor of functions. Similarly, $\forall$ is used both as a quantifier and to specify dependent types. When we see `a` $\to$ `b`, without further context, we cannot tell whether we have the type of a function with domain `a` and codomain `b` or the proposition "`a` implies `b`." Similarly, $\forall$`x, t[x]` can denote a proposition or a dependent type.

It turns out that not only the two pairs of concepts look the same, they *are* the same. This is called the *Curry–Howard correspondence*. It is also called the *PAT principle*, where PAT is a double mnemonic:

PAT = propositions as types PAT = proofs as terms

Nicolaas Govert de Bruijn, Haskell Curry, William Alvin Howard, and possibly others independently noticed that for some logics, propositions are isomorphic to types, and proofs are isomorphic to terms. Hence, in dependent type theory, we *identify* proofs with terms as well as propositions with types, a considerable economy of concepts.

Let us go through the dramatis personae one by one: types, propositions, terms, and proofs. We use the metavariables σ, τ for types; φ, ψ for propositions; `t`, `u`, `x` for terms; and `h` for proofs.

For types, we have the following:

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

For propositions, we have the following:

- $\varphi \to \psi$ can be read as "φ implies ψ", or as the type of functions mapping proofs of φ to proofs of ψ.
- $\forall$`x` $:\sigma$`, `ψ`[x]` can be read as "for all x, ψ`[x]`", or as the type of function mapping values `x` of type σ to proofs of ψ`[x]`.
- $\forall$`h` $:\varphi$`, `ψ`[h]` is the type of function mapping proofs `h` of φ to proofs of ψ`[h]`.

Notice the similarities between types and propositions. And recall that $\forall$ is an alias for Π.

For terms, we have the following:

- Declared constants are terms.
- `t u` is the application of function `t` to value `u`.
- λ`x, t[x]` is a function mapping `x` to `t[x]`.

For proofs (i.e., proof terms), we have the following:

- A lemma name is a proof.
- `l a`, which instantiates the leading parameter or quantifier of lemma `l` with value `a`, is a proof.
- `l l'`, which discharges the leading assumption of `l` with lemma `l'`, is a proof. This operation is called *modus ponens*.

- Given `h[n]` : φ`[n]`, we have that λ`n` : σ, `h[n]` is a proof of $\forall$`n` : σ, φ`[n]`.
- Given `h[h']` : φ`[h']`, we have that λ`h'` : ψ, `h[h']` is a proof of $\forall$`h'` : ψ, φ`[h']`.

For the last two cases, in a structured proof we would write `assume`, and in a tactical proof we would invoke `intros`.

By the Curry–Howard correspondence, a proof by *induction* is the same as a *recursively* specified proof term. When we invoke the induction hypothesis, we are really just calling a recursive function (the lemma) recursively. This explains why, in a proof by pattern matching (Section 3.3), the induction hypothesis has the same name as the lemma we prove.

In the same way, *well-founded induction* (also called "complete induction", "course-of-values induction", or "strong induction") is the same as well-founded recursion. *Structural induction* corresponds to *structural recursion*. Lean's termination checker is used to prove well-foundedness of the proof by induction.

Finally, you may have noticed that the commands `def` and `lemma` have almost the same syntax. They are, in fact, almost the same, with the exception that `lemma` considers the proof term (or whatever defined term) opaque, whereas `def` keeps it transparent. Since proofs are irrelevant to Lean's logic, there is no need to store them or unfold them later. A similar difference is visible between `let` and `have`.

3.6 Summary of New Syntax

Proof Commands

`assume`	states variables or assumptions
`calc`	combines proofs by transitivity
`have`	states an intermediate lemma
`let ... in`	introduces a local definition
`show`	states the goal's conclusion

Tactics

`have`	states an intermediate lemma
`let`	introduces a local definition

Part II

Functional–Logic Programming

Chapter 4

Functional Programming

We take a closer look at the basics of typed functional programming: inductive types, pattern matching, recursive functions, and proofs by induction. The concepts covered here are described in more detail in Chapters 7 to 9 of *Theorem Proving in Lean* [1].

4.1 Inductive Types

Inductive types are modeled after the datatypes of typed functional programming languages (e.g., Haskell, ML, OCaml). As we saw in Chapter 1.2, 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 or ε that cannot be expressed using a finite combination of `zero` and `succ`, and "no confusion" ensures that `zero` $\neq$ `succ n` for all `n`. In addition, inductive types are always finite. The infinite term `succ (succ (succ ...))` is not a value, nor does there exist a number `n` such that `n = succ n`.

Already in Chapter 1, 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 some new types.

4.2 Example: Lists

We first define the reverse of a list, exploiting the predefined concatenation operator ++:

```
def reverse {α : Type} : list α → list α
| []        := []
| (x :: xs) := reverse xs ++ [x]
```

(In fact, `reverse` exists in Lean's standard library, as `list.reverse`, but our definition is more suitable for reasoning.) A nice 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. For the induction step, the goal to prove is

```
reverse (reverse xs ++ [x]) = x :: xs
```

and the induction hypothesis is

```
∀xs : list α, reverse (reverse xs) = xs
```

We need a way to "distribute" `reverse` over `++`, to eliminate the occurrence of `++ [x]` and obtain a term that matches the induction hypothesis's left-hand side. The trick is to prove and use the following lemma:

```
lemma 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]
```

It is important to state the lemma generally, for all `xs` and `ys`. If we put `[y]` instead of `ys`, we would run into difficulties when proving the lemma's induction step. In general, finding the right inductions and lemmas can be a difficult task, which requires thought and creativity.

With the `reverse_append` lemma in place, we can prove our initial goal by a straightforward induction:

```
lemma reverse_reverse {α : Type} :
  ∀xs : list α, reverse (reverse xs) = xs
| []        := by refl
| (x :: xs) :=
  by simp [reverse, reverse_append, reverse_reverse xs]
```

The next operation we define is a *map function*: 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 xs
```

Notice that because `f` does not change in the recursive call, we put it as a parameter of the entire definition. This means we must write `map xs` and not `map f xs` in the recursive call. However, when using the function later, we would need to pass an argument for `f`. The alternative, which is the only option for arguments that change in recursive calls, would be as follows:

```
def map₂ {α β : Type} : (α → β) → list α → list β
| _ []        := []
| f (x :: xs) := f x :: map₂ f xs
```

Notice the use of `_` as a nameless placeholder in the first case. We could also have given the variable a name (e.g., `f`).

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

```
lemma map_ident {α : Type} :
  ∀xs : list α, map (λx, x) xs = xs
| []        := by refl
| (x :: xs) := by simp [map, map_ident xs]
```

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

```
lemma map_comp {α β γ : Type} (f : α → β) (g : β → γ) :
  ∀xs : list α, map (λx, g (f x)) xs = map g (map f xs)
| []        := by refl
| (x :: xs) := by simp [map, map_comp xs]
```

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

```
lemma map_append {α β : Type} (f : α → β) :
  ∀xs ys : list α, map f (xs ++ ys) = map f xs ++ map f ys
```

```
| []        ys := by refl
| (x :: xs) ys := by simp [map, map_append xs]

lemma map_reverse {α β : Type} (f : α → β) :
  ∀xs : list α, map f (reverse xs) = reverse (map f xs)
| []        := by refl
| (x :: xs) :=
  by simp [map, reverse, map_append, map_reverse xs]
```

The length of a list is defined by recursion:

```
def length {α : Type} : list α → ℕ
| []        := 0
| (x :: xs) := length xs + 1
```

The next 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
  | tt := bcount xs + 1
  | ff := bcount xs
  end
```

The `bcount` function counts the number of elements in a list that satisfy the given predicate `p`. The predicate's codomain is `bool`, not `prop`. As a general rule, we will use type `bool`, of Booleans, when specifying programs and use the type `prop`, of propositions, when stating properties of programs. The two values of type `bool` are called `tt` and `ff`. The connectives are called `bor` (infix: `||`), `band` (infix: `&&`), `bnot`, and `bxor`.

The `match` construct has the following general syntax:

```
match term₁, ..., termₘ with
| patterns₁₁, ..., pattern₁ₘ := result₁
    ⋮
| patternsₙ₁, ..., patternₙₘ := resultₙ
end
```

Notice the differences with the pattern matching syntax in `def` and `lemma`: The patterns are separated by commas and not by spaces. This reduces the number of required parentheses. The patterns may contain variables, constructors, and _ placeholders. The $result_i$ expressions may refer to the variables introduced in the corresponding patterns.

The next list operation extracts the first element of a list:

```
def head {α : Type} (xs : list α) : option α :=
match xs with
| []     := none
| x :: _ := some x
end
```

Because an empty list contains no elements, there is no meaningful value we can return in that case. For this reason, we use an `option`. The type `option` α has two constructors: `none` : `option` α and `some` : $\alpha \to$ `option` α. We use `none` when we have no meaningful value to return and `some` otherwise.

Given two lists of the same length [x_1, ..., x_n] and [y_1, ..., y_n], the zip operation constructs a list of pairs [(x_1, y_1), ..., (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)]`.

Pairs are written using traditional notations. The type constructor $\times$, which occurs in the result type `list (α × β)`, has an infix syntax. The pair constructor is written using the syntax `(t, u)`, where `t` and `u` are the arguments. Given a pair `p`, the first and second components can be extracted by writing `p.fst` and `p.snd` or `p.1` and `p.2`.

The `p.1` and `p.2` syntaxes are used below to state a distributivity law about `map` and `zip`:

```
lemma map_zip {α α' β β' : Type} (f : α → α') (g : β → β') :
  ∀xs ys, map (λp : α × β, (f p.1, g p.2)) (zip xs ys) =
    zip (map f xs) (map g ys)
| (x :: xs) (y :: ys) :=
  begin
    simp [zip, map],
    exact (map_zip _ _)
  end
| []        _         := by refl
| (_ :: _)  []        := by refl
```

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

```
lemma length_zip {α β : Type} :
  ∀(xs : list α) (ys : list β),
    length (zip xs ys) = min (length xs) (length ys)
| (x :: xs) (y :: ys) :=
  by simp [zip, length, length_zip xs ys, min_add_add]
| []        _         := by refl
| (_ :: _)  []        := by refl
```

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

```
lemma min_add_add (l m n : ℕ) :
  min (m + l) (n + l) = min m n + l :=
have l + m ≤ l + n ↔ m ≤ n :=
begin
  rw add_comm l,
  rw add_comm l,
  apply nat.add_le_add_iff_le_right
end,
by by_cases m ≤ n; simp [min, *]
```

The `min` function is defined such that `min a b = (if a ≤ b then a else b)`. To reason about it, we often need to perform a case analysis on the condition `a ≤ b`. This is achieved using `by_cases`. Given a condition `c`, it creates two subgoals: one in which `c` appears as a hypothesis and one where `¬ c` appears.

In the next example, we show how to exploit injectivity of constructors. The `cases` tactic can be used to apply injectivity on equations whose both sides have the same constructor applied. In the proof below, the equation on which injectivity is applied is `x :: xs = y :: ys`. As a result of the case distinction, `y` is replaced by `x` and `ys` is replaced by `xs` throughout the goal:

```
lemma injection_example {α : Type} (x y : α) (xs ys : list α)
    (h : list.cons x xs = list.cons y ys) :
  x = y ∧ xs = ys :=
```

```
begin
  cases h,
  apply and.intro,
  { refl },
  { refl }
end
```

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

```
lemma distinctness_example {α : Type} (x y : α) (xs ys : list α)
    (h : [] = y :: ys) :
  false :=
by cases h
```

4.3 Example: Binary Trees

Lists are not the only interesting data type. 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
| empty {} : btree
| node     : α → btree → btree → btree
```

(The `{}` annotation is often used with nullary constructors of polymorphic types. Here, it indicates that the type α should be implicitly derived from the context surrounding occurrences of `empty`.)

By default, the constructor names are prefixed by the name of the type—i.e., `tree.empty` and `tree.node`. To lighten notations, we can enter the following command, which exports `empty` and `node` from the `tree` namespace, allowing us to write `empty` and `node` with no qualification:

```
export btree (empty node)
```

A peculiarity of binary trees is that structural induction has two induction hypotheses: one for the left subtree of an inner node and one for the right subtree. To prove a goal `p t` about a tree `t` by structural induction on `t`, we must prove (1) `p empty` and (2) `p (node a l r)` assuming `p l` and `p r`.

The tree counterpart to list reversal is the mirror operation:

```
def mirror {α : Type} : btree α → btree α
| empty        := empty
| (node a l r) := 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 the example below demonstrates:

```
lemma mirror_mirror {α : Type} :
  ∀t : btree α, mirror (mirror t) = t
| empty        := by refl
| (node a l r) :=
  begin
    simp [mirror],
    apply and.intro,
    { apply mirror_mirror l },
    { apply mirror_mirror r }
  end
```

4.4 Example: Colors

Lean provides a convenient syntax for defining records, or *structures*. These are essentially nonrecursive, single-constructor inductive types, but with some syntactic sugar. The definition below introduces an `rgb` structure with three fields of type $\mathbb{N}$ called `red`, `green`, and `blue`:

```
structure rgb :=
(red green blue : ℕ)
```

We can define a new structure as the extension of an existing structure. The definition below introduces a fourth component, called `alpha`, to `rgb` and calls the result `rgba`:

```
structure rgba extends rgb :=
(alpha : ℕ)
```

Values can be specified in a variety of syntaxes:

```
def red : rgb := { red := 0xff, green := 0x00, blue := 0x00 }
def semitransparent_red : rgba := { alpha := 0x7f, ..red }
```

The definition of `semitransparent_red` copies all the values from `red` (indicated by the `..red` syntax) except for the `alpha` field, which it overrides.

Next, we define an operation called `shuffle`:

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

The definition uses the automatically generated projection functions `color.red`, `color.green`, and `color.blue`. Instead of `color.red c`, Lean lets us simply write `c.red`, and similarly for the other fields.

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

```
lemma shuffle_shuffle_shuffle (c : rgb) :
  shuffle (shuffle (shuffle c)) = c :=
by cases c; refl
```

The proof teaches us a new trick. The `cases` tactic is a relative of `induction`. It performs a case distinction on its arguments, but it does not generate inductive hypotheses.

4.5 Overlapping Patterns

It is possible to write equations with overlapping patterns. For example:

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

The first matching equation is taken. Internally, this is the same as

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

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

4.6 New Tactics

by_cases

```
by_cases proposition
```

The `by_cases` tactic performs a case analysis on a proposition. It is useful to reason about the condition in an `if` condition.

cases

cases *variable* [with *name*$_1$ ... *name*$_n$]

The `cases` tactic performs a case distinction on the specified variable. This gives rise to as many subgoals as there are constructors in the definition of the variable's type. Its behavior is similar to `induction`'s except that it does not produce induction hypotheses.

The optional names *name*$_1$, ..., *name*$_n$ are used for any emerging hypotheses or local constants, to override the default names.

4.7 Summary of New Syntax

Declaration

`structure`	introduces a record type and its projection functions

Command

`export`	makes names available outside their namespaces

Term Language

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

Tactics

`by_cases`	performs a case analysis on an `if` condition
`cases`	performs a case analysis on a variable of an inductive type

Chapter 5

Inductive Predicates

Inductive predicates are reminiscent of the Horn clauses of Prolog-style logic programming languages. But Lean offers a much stronger logic than Prolog, so we need to do some work—interactive proving—to establish theorems instead of just running the Prolog interpreter. A possible view of Lean:

Lean = typed functional programming + logic programming + more logic

5.1 Introductory Example

Inductive predicates, or (more precisely) inductively defined propositions, are familiar from mathematics. Consider the example:

> The set E of even natural numbers is defined as the smallest set closed under the following rules: (1) $0 \in E$; and (2) for every $n \in \mathbb{N}$, if $n \in E$, then $n + 2 \in E$.

In Lean, we write

```
inductive even : ℕ → Prop
| zero    : even 0
| add_two : ∀n, even n → even (n + 2)
```

This should look familiar: By the Curry–Howard correspondence, what we have done is introduce a new unary type constructor, `even`. Inductive types and inductive predicates are implemented as the same mechanism in Lean.

The `even` predicate has two constructors: `zero` and `add_two`. These can be used to build proof terms. The `even` predicate can be seen as a tree type, the trees being the corresponding proof terms (or proof trees). Thanks to the "no junk" guarantee of inductive definitions, `zero` and `add_two` are the only two ways to construct `even`. Thus, there is no danger of accidentally proving that `1` is even.

5.2 Logical Symbols

The truth values `false` and `true`, the connectives $\wedge$ and $\vee$, the $\exists$ quantifier, and the equality predicate = are all defined as inductive propositions or predicates:

```
inductive false : Prop

inductive true : Prop
| intro : true

inductive and (a b : Prop) : Prop
```

```
| intro : a → b → and

inductive or (a b : Prop) : Prop
| intro_left : a → or
| intro_right : b → or

inductive Exists {α : Type} (p : α → Prop) : Prop
| intro : ∀a : α, p a → Exists

inductive eq {α : Type} : α → α → Prop
| refl (a : α) : eq a a
```

The notations ∃x : α, p and x = y are syntactic sugar for `Exists (λx : α, p)` and `eq x y`, respectively. In contrast, ∀ (= Π) and → are built directly into the logic and are not defined as inductive predicates. Notice that there is no constructor for `false`. There are no proofs of `false`, just like there is no proof that `1` is even.

5.3 Introduction, Elimination, Induction, and Inversion

We saw in Section 2.3 that the logical connectives and the ∃ quantifier are equipped with introduction and elimination rules. The same is true for arbitrary inductive predicates `p`.

`p`'s constructors are introduction rules. They typically have the form ∀..., ... → p .... They can be used to prove goals of the form `p` ....

Elimination works the other way around. It extracts information from a lemma or hypothesis of the form `p` .... Elimination takes various forms: pattern matching (at the top-level of a definition or lemma, or with `match`), the `cases` and `induction` tactics, or custom elimination rules (e.g., `and.elim_left`).

Just as we can perform induction on a term, we can perform induction on a proof term using the `induction` tactic. This is sometimes called *rule induction*, because the induction is on the introduction rules (i.e., the constructors of the proof term).

Often it is convenient to rewrite concrete terms of the form `p (c ...)`, where `c` is typically a constructor. We can state and prove an *inversion rule* to support such eliminative reasoning. The general format of an inversion rule (assuming a binary constructor) is as follows:

∀x y, p (c x y) → (∃..., ... ∧ ...) ∨ ... ∨ (∃..., ... ∧ ...)

It can be useful to combine introduction and elimination in a single lemma, which can be used for rewriting both the hypotheses and the conclusions of goals:

∀x y, p (c x y) ↔ (∃..., ... ∧ ...) ∨ ... ∨ (∃..., ... ∧ ...)

For example:

∀n : ℕ, even n ↔ n = 0 ∨ ∃m : ℕ, n = m + 2 ∧ even m

5.4 Example: Full Binary Trees

We now review in turn four examples of inductive predicates. Our first example is based on the type of binary trees introduced in Section 4.3:

A binary tree is *full* if all its nodes have either zero or two children. This can be encoded as an inductive predicate as follows:

```
inductive is_full {α : Type} : btree α → Prop
| empty : is_full empty
| node (a : α) (l r : btree α) (hl : is_full l) (hr : is_full r)
    (empty_iff : l = empty ↔ r = empty) :
  is_full (node a l r)
```

The first case states that the empty tree is a full tree. The second case states that a tree is a full tree if has two child trees that are themselves full and that either both are empty or both are nonempty.

Syntactically, we can put the arguments on the left of the colon, as parameters, or on the right. Here is an equivalent definition of the latter kind:

```
inductive is_full₂ {α : Type} : btree α → Prop
| empty : is_full₂ empty
| node : ∀(a : α) (l r : btree α), is_full₂ l → is_full₂ r →
    (l = empty ↔ r = empty) →
  is_full₂ (node a l r)
```

The tree that consists of a node with empty tree as children is a full tree. Here is a simple proof, using the introduction rule `is_full.node`:

```
lemma is_full_singleton {α : Type} (a : α) :
  is_full (node a empty empty) :=
begin
  apply is_full.node,
  repeat { apply is_full.empty },
  refl
end
```

The proof makes use of the `repeat` combinator, which applies the tactic specified within curly braces repeatedly, as long as it succeeds. Here it is applied twice, to eliminate the two identical goals `is_full empty`.

A somewhat more interesting property of full trees is that fullness is preserved by the mirror operation. Our first proof is by structural induction on `t`:

```
lemma is_full_mirror {α : Type} :
  ∀t : btree α, is_full t → is_full (mirror t)
| empty        := by intro; assumption
| (node a l r) :=
  begin
    intro full_t,
    cases full_t,
    rw mirror,
    apply is_full.node,
    repeat { apply is_full_mirror, assumption },
    simp [mirror_eq_empty_iff, *]
  end
```

The key to the proof is the `cases` tactic invocation on the hypothesis `full_t : is_full (node a l r)`. The tactic notices, via a simple syntactic check, that the first introduction rule (`is_full.empty`) cannot have been used to derive `is_full (node a l r)`, so it only produces one case, corresponding to the second introduction rule (`is_full.node`). As usual, the tactical proof makes more sense if you inspect it in Visual Studio Code, moving the cursor around.

Another way to carry out the proof is to perform pattern matching on `is_full t` at the top level:

```
lemma is_full_mirror₂ {α : Type} :
  ∀t : btree α, is_full t → is_full (mirror t)
| _ is_full.empty :=
  begin
    rw mirror,
    exact is_full.empty
  end
| _ (is_full.node a l r hl hr empty_iff) :=
```

```
  begin
    rw mirror,
    apply is_full.node,
    repeat { apply is_full_mirror₂, assumption },
    simp [mirror_eq_empty_iff, *]
  end
```

A third approach relies on an inversion rule. The rule is proved by a combination of introduction and elimination steps:

```
lemma is_full_node_iff {α : Type} (a : α) (l r : btree α) :
  is_full (node a l r) ↔
  is_full l ∧ is_full r ∧ (l = empty ↔ r = empty) :=
begin
  apply iff.intro,
  { intro h,
    cases h,
    cc },
  { intro h,
    apply is_full.node,
    repeat { cc } }
end
```

The proof relies on the `cc` tactic, which implements an algorithm called congruence closure. The tactic studies the equalities present in the problem and attempts to derive a proof from them. It is described in slightly more detail below (Section 5.9).

Equipped with the inversion rule, we can perform a simpler proof that relies on the `simp` tactic:

```
lemma is_full_mirror₃ {α : Type} :
  ∀t : btree α, is_full t → is_full (mirror t)
| _ is_full.empty                          :=
  by rw mirror; exact is_full.empty
| _ (is_full.node a l r hl hr empty_iff) :=
  by simp [mirror, is_full_node_iff, is_full_mirror₃ l hl,
    is_full_mirror₃ r hr, mirror_eq_empty_iff, empty_iff]
```

5.5 Example: Sorted Lists

Our next example is a predicate that checks whether a list of natural numbers is sorted in increasing order:

```
inductive sorted : list ℕ → Prop
| nil : sorted []
| single {x : ℕ} : sorted [x]
| two_or_more {x y : ℕ} {xs : list ℕ} (xy : x ≤ y)
    (yxs : sorted (y :: xs)) :
  sorted (x :: y :: xs)
```

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:

```
example :
  sorted [3, 5] :=
begin
  apply sorted.two_or_more,
  { linarith },
  { exact sorted.single }
```

```
end
```

The example needs two of the introduction rules. Notice the invocation of `linarith`. This tactic can be invoked to solve linear arithmetic goals. It is described below (Section 5.9). A more compact proof follows, using proof terms for everything except the arithmetic subgoal:

```
example :
  sorted [3, 5] :=
sorted.two_or_more (by linarith) sorted.single
```

The same idea can be used to prove that `[7, 9, 9, 11]` is sorted:

```
example :
  sorted [7, 9, 9, 11] :=
sorted.two_or_more (by linarith)
  (sorted.two_or_more (by linarith)
    (sorted.two_or_more (by linarith)
      sorted.single))
```

Conversely, we can show that some lists are not sorted, but this time we need to use elimination, in the form of pattern matching:

```
example :
  ¬ sorted [17, 13] :=
assume h : sorted [17, 13],
have 17 ≤ 13 :=
  match h with
  | sorted.two_or_more xy yxs := xy
  end,
have ¬ (17 ≤ 13) := by linarith,
show false, from by cc
```

To prove a negation, we assume the unnegated formula and prove false. From the assumption that `[17, 13]` is sorted, we derive that `17 ≤ 13`, which together with the obvious numeric truth that `¬ (17 ≤ 13)` leads to a contradiction. The contradiction is easily detected by `cc`.

5.6 Example: Well-formed and Ground First-Order Terms

Our third example is based on an inductive type of first-order terms:

```
inductive term (α β : Type) : Type
| var {} : β → term
| fn     : α → list term → term

export term (var fn)
```

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 metavariables t_j stand for the arguments, which are themselves terms. The parameters α and β are the types of function symbols and variables, respectively.

Not all terms are legal. For example, the term $g(f(a), f(a,b))$ is considered ill formed, because the function f is invoked with different number of arguments (once with one argument and once with two). Along with α and β, we also consider the arity, represented by a function `arity` : $\alpha \to$ `N` indicating how many arguments each function symbol takes. The `well_formed` predicate then checks whether the given term only contains function symbol applications with the specified number of arguments:

```
inductive well_formed {α β : Type} (arity : α → ℕ) :
  term α β → Prop
| var (x : β) : well_formed (var x)
| fn (f : α) (ts : list (term α β))
    (hargs : ∀t ∈ ts, well_formed t)
    (hlen : list.length ts = arity f) :
  well_formed (fn f ts)
```

The `fn` case checks that the arguments `ts` are recursively well formed. Additionally, it requires 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 variable_free {α β : Type} : term α β → Prop
| fn (f : α) (ts : list (term α β))
    (hargs : ∀t ∈ ts, variable_free t) :
  variable_free (fn f ts)
```

This time, there is no `var` case, for obvious reasons.

5.7 Example: Reflexive Transitive Closure

Our last example is the reflexive transitive closure of a relation, modeled as a binary predicate `rtc`.

```
inductive rtc {α : Type} (r : α → α → Prop) : α → α → Prop
| base (a b : α) : r a b → rtc a b
| refl (a : α) : rtc a a
| trans (a b c : α) : rtc a b → rtc b c → rtc a c
```

One of the key properties of `rtc` is that it is idempotent—applying `rtc` to `rtc r` has no effect, for any binary predicate `r`:

```
lemma rtc_rtc_iff_rtc {α : Type} (r : α → α → Prop) (a b : α) :
  rtc (rtc r) a b ↔ rtc r a b :=
begin
  apply iff.intro,
  { intro h,
    induction h,
    case rtc.base : x y {
      assumption },
    case rtc.refl : x {
      apply rtc.refl },
    case rtc.trans : x y z {
      apply rtc.trans,
      assumption,
      assumption } },
  { intro h,
    apply rtc.base,
    assumption }
end
```

In the induction proof, we use the `case` syntax both to increase readability of the proof script and to give intuitive names to the emerging variables. It is easy to get lost in goals containing long, automatically generated names. The goal management tactics discussed in Section 2.7 can also be a great help when facing large goals.

We can state the idempotence property more standardly in terms of equality instead of as an equivalence:

```
lemma rtc_rtc_eq_rtc {α : Type} (r : α → α → Prop) :
  rtc (rtc r) = rtc r :=
begin
  apply funext,
  intro a,
  apply funext,
  intro b,
  apply propext,
  apply rtc_rtc_iff_rtc
end
```

The proof requires two lemmas that are available because Lean's logic is classical. *Functional extensionality* (`funext`) states that two functions that yield equal results for all inputs must be equal. *Propositional extensionality* (`propext`) states that equivalence of propositions coincides with equality.

```
funext            : (∀x, ?f x = ?g x) → ?f = ?g
propext           : (?a ↔ ?b) → ?a = ?b
```

These properties may seem obvious, but there are proof assistants built on weaker, intuitionistic logics in which the property do not generally hold.

5.8 Induction Pitfalls

Inductive predicates often have parameters that evolve through the induction. Pattern matching and `cases` handle this gracefully, but some care is necessary when invoking `induction`. Such details are often glossed over in informal proofs, but proof assistants require us to be precise.

Suppose we have the definition

```
inductive even : ℕ → Prop
| zero    : even 0
| add_two : ∀n, even n → even (n + 2)
```

If the goal has the form `even n → p n`, applying `induction` will produce the following cases:

- `p 0`
- `∀n, p n → p (n + 2)`

This works as desired.

The problem is that when the argument to `even` is not a variable, `induction` destroys the identity of the argument. (With a variable, there is nothing to destroy.) Thus, applying `induction` on `even (2 * n + 1) → false` produces the subgoals

- `false`
- `false → false`

The first subgoal is not provable. To solve this issue, we need to replace `2 * n + 1` by a variable `x` and add an equation `x = 2 * n + 1` as a hypothesis:

```
∀x, even x → x = 2 * n + 1 → false
```

This proposition is logically equivalent, but now we obtain very different subgoals:

- `2 * n + 1 = 0 → false`
- `∀x, (2 * n + 1 = x → false) → 2 * n + 1 = x + 2 → false`

Sadly, the second subgoal is not provable. The problem is that we want to instantiate the `n` in the assumption with `n + 1` but `n` is not instantiable.

The solution? We need to explicitly quantify over `n` to be able to instantiate it in the induction hypothesis. We state our goal as

$$\forall \texttt{x, even x} \to \forall \texttt{n, x = 2 * n + 1} \to \texttt{false}$$

Now we obtain the subgoals

- $\forall \texttt{n, 2 * n + 1 = 0} \to \texttt{false}$;
- $\forall \texttt{x, (}\forall \texttt{n, 2 * n + 1 = x} \to \texttt{false)} \to \forall \texttt{n, 2 * n + 1 = x + 2} \to \texttt{false}$.

Now the variable `n` in the conclusion is disconnected from the variable `n` in the assumption, and we can instantiate the assumption's `n` with the conclusion's `n + 1`.

5.9 New Tactics

cc

The `cc` tactic applies an algorithm known as *congruence closure* to derive new equalities from those that are present in the goal. For example, if the goal contains `b = a` and `f b` $\neq$ `f a` as hypotheses, the congruence closure procedure will derive `f b = f a` from `b = a` and discover a contradiction with `f b` $\neq$ `f a`. The tactic is also suitable for more pedestrian work, inferring for example `a` $\vee$ `b` $\vee$ `c` from `b`.

linarith

The `linarith` tactic can be used to prove goals involving linear arithmetic equalities or inequalities. "Linear" means that multiplication and division do not occur, of if they do they one of the operand must be a numeric constant. For example, `2 * x < y` is a linear constraint—indeed, it can be rewritten as `x + x < y`—whereas `x * y < y` is nonlinear.

norm_num

The `norm_num` tactic can be used to prove goals requiring normalization of arithmetic expressions. For example, it can be used to prove that `1 + (x + -1) = x`.

5.10 Summary of New Syntax

Tactic Combinators

`repeat`	repeatedly applies a tactic on all goals until failure

Tactic

`cc`	propagates the equalities in the goal
`linarith`	applies a procedure for linear arithmetic
`norm_num`	normalizes arithmetic expressions

Chapter 6

Monads

We take a look at an important functional programming abstraction: monads. Monads generalize computation with side effects. Haskell has shown that monads can be used very successful to write imperative programs. In Lean, they are used to write imperative programs, but also to reason about them via the so-called monadic laws. Monads are even useful for programming Lean itself—or metaprogramming Lean—as we will see in Chapter 7.

These notes are inspired by Chapter 7 of *Programming in Lean* [2]. We strongly recommend that you read that chapter to get a more complete picture of monads in Lean. We also refer to Chapter 14 of *Real World Haskell* [18] for a general introduction to monads.

6.1 Motivating Example

Consider the following programming task: Implement a function `sum_2_5_7 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 if the list has fewer than seven elements, you can return `none`.

A straightforward implementation follows:

```
def sum_2_5_7₁ (ns : list ℕ) : option ℕ :=
match list.nth ns 1 with
| none    := none
| some n2 :=
  match list.nth ns 4 with
  | none    := none
  | some n5 :=
    match list.nth ns 6 with
    | none    := none
    | some n7 := some (n2 + n5 + n7)
    end
  end
end
```

(Recall that `nth` counts elements from 0; thus, the second element is at position 1, etc.) The code is quite ugly, because of all the pattern matching on `option`. Can we do better? It turns out we can. We can put all the ugliness in one function, which we call `bind_option`:

```
def bind_option {α : Type} {β : Type} :
  option α → (α → option β) → option β
| none     f := none
| (some a) f := f a
```

That function operates on an option (a value of type `option`). If the option is `none`, we leave it as is. This corresponds to an error condition, and errors are preserved. Otherwise, if the option is `some a` for some `a`, then we apply the specified operation `f` on `a`.

We can now use `bind_option` to program our sum function:

```
def sum_2_5_7₂ (ns : list ℕ) : option ℕ :=
bind_option (list.nth ns 1)
  (λn2, bind_option (list.nth ns 4)
    (λn5, bind_option (list.nth ns 6)
      (λn7, some (n2 + n5 + n7))))
```

Intuitively, the program reads as follows:

- Extract the second item from the list. If it is `none`, we are done. Otherwise, we obtain this item as `n2` and continue with the next step.
- Perform the same for the fifth and seventh item, mutatis mutandis.
- Return the sum of `n2`, `n5`, and `n7` in a `some` wrapper.

Semantically, our new function `sum_2_5_7₂` is equal to the original `sum_2_5_7`.

Instead of defining `bind_option` 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 sum_2_5_7₃ (ns : list ℕ) : option ℕ :=
bind (list.nth ns 1)
  (λn2, bind (list.nth ns 4)
    (λn5, bind (list.nth ns 6)
      (λn7, some (n2 + n5 + n7))))
```

One of the advantages of using the predefined notion is that it provides syntactic sugar, in the form of an infix >>= operator:

```
def sum_2_5_7₄ (ns : list ℕ) : option ℕ :=
list.nth ns 1 >>=
  λn2, list.nth ns 4 >>=
    λn5, list.nth ns 6 >>=
      λn7, some (n2 + n5 + n7)
```

The syntax `ma >>= f` expands to `bind ma f`.

The penultimate version of the sum program uses heavier syntactic sugar:

```
def sum_2_5_7₅ (ns : list ℕ) : option ℕ :=
  do n2 ← list.nth ns 1,
    do n5 ← list.nth ns 4,
      do n7 ← list.nth ns 6,
        some (n2 + n5 + n7)
```

The `do` notation provides an intuitive notation for monads. `do a ← ma, t` is syntactic sugar for `ma >>= λa, t`. If we do not care about the return result of `ma`'s computation, we can write `do ma, t`, which expands to `ma >>= λ_, t`.

For convenience, the notation allows multiple ← assignments in a single `do` block. This brings us to the final version of the program:

```
def sum_2_5_7₆ (ns : list ℕ) : option ℕ :=
do
  n2 ← list.nth ns 1,
  n5 ← list.nth ns 4,
  n7 ← list.nth ns 6,
  some (n2 + n5 + n7)
```

Each line with an arrow ← attempts to read a value. In case of failure, the entire program evaluates to `none`. Although the notation has an imperative flavor, the function is a pure functional program.

6.2 Overview

The `option` type constructor is an example of a monad. In general, a monad is a type constructor `m` that depends on some type parameter α—i.e., `m` α—equipped with two distinguished operations:

$$\texttt{pure} : \Pi\{\alpha : \texttt{Type}\},\ \alpha \to \texttt{m}\ \alpha$$
$$\texttt{bind} : \Pi\{\alpha\ \beta : \texttt{Type}\},\ \texttt{m}\ \alpha \to (\alpha \to \texttt{m}\ \beta) \to \texttt{m}\ \beta$$

Recall that curly braces denote implicit arguments. For options, the `pure` operation is simply `some`, whereas `bind` is what we called `bind_option`.

Intuitively, we can think of a monad as a "box" containing some data. The operation `pure` puts the 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 is an `m` β monad, not an `m` α monad. But there is no general way to extract the data from the monad, i.e., to obtain an α from an `m` α.

To summarize, `pure a` (also called `return a`) simply provides a box containing the the value `a`, with no side effects, whereas `bind ma f` (also written `ma >>= f`) executes `ma`, then executes `f` with the boxed result `a` of `ma`.

Monads are an abstract concept with many applications. The option monad is only one instance among many. The following table gives an overview of monad instances and the effects they provide.

Type	Effect
`option` α	failure when `none`; otherwise, `some (a :` α`)` is the result
`set` α	nondeterministic behavior (set of behaviors)
`t` $\to \alpha$	reading elements of type `t` (e.g., a configuration)
$\mathbb{N} \times \alpha$	adjoining running time (e.g., to model algorithmic complexity)
`string` $\times\ \alpha$	adjoining text output (e.g., for logging)
`prob` α	probability (e.g., using random number generators)
`io` α	interaction with the operating system
`tactic` α	interaction with the prover

All of the above are type constructors `m` are parameterized by a type α. Some effects can be combined (e.g., `option (t` $\to \alpha$`)`). Some effects are not executable (e.g., `set` α, `prob` α); they are nonetheless useful for modeling programs abstractly in the logic.

Specific monads `m` may provide further operators. For example, they may provide a way to extract the boxed value stored in the monad without `bind`'s requirement of putting it back in a monad.

Monads have several benefits. We get the convenient and highly readable `do` notation. They also offer generic operations, such as `mmap :` $(\alpha \to$ `M` $\beta) \to$ `list` $\alpha \to$ `M (list` β`)`, which work uniformly across all monad instances. To quote *Programming in Lean* [2], "The power of the abstraction is not only that it provides general functions and notation the 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."

6.3 Monadic Laws

The `bind` and `pure` operations are normally required to obey three laws, called the monadic laws.

The `bind` operation combines two programs. If either of these is simply pure data, we can eliminate the `bind`. This gives us the first two monadic laws. The first law:

```
(do                                  =     f a
  a' ← pure a,
  f a')
```

The second law:

```
  (do                                =     x
    a ← x,
    pure a)
```

The third law is an associativity rule for bind. It allows us to flatten a nested computation:

```
(do                                  =     (do
  b ← (do                                    a ← x,
    a ← x,                                   b ← f a,
    f a),                                    g b)
  g b)
```

6.4 A Type Class of Monads

Monads are a mathematical structure, so we use `class` to add them as a type class. Type classes will be covered in more detail in Chapter 12. We can think of a type class as a structure or record that is parameterized by a type—or here, by a type constructor `m : Type → Type`.

A possible Lean definition of monads, together with the three monadic laws, follows:

```
class lawful_monad (m : Type → Type)
  extends has_bind m, has_pure m : Type 1 :=
(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 >>= (λa, f a >>= g)))
```

Let us analyze this definition step by step:

- We are creating a structure parameterized by a unary type constructor `m : Type → Type`.
- The type class inherits the fields, and any syntactic sugar, from type classes called `has_bind` and `has_pure`. These provide the `bind` and `pure` operations on `m`, with the expected types and the syntactic sugar.
- The type `Type 1` is a technical necessity. Because monads quantify over types, they cannot be standard types themselves. They need to live in a larger type universe, called `Type 1`. (The familiar `Type` type of types is also called `Type 0`.) We will come back to this fine point in Chapter 11.
- Finally, the definition adds three fields to those already provided by `has_bind` and `has_pure`. Each field is a proof of one of the three monadic laws.

We call our type class "lawful monad" because the monadic laws are required to hold. To instantiate this definition with a concrete monad (e.g., the option monad), we must supply the monad `m`, suitable `bind` and `pure` operators, and proofs corresponding to the three monadic laws.

Our definition captures the essence of Lean's predefined concept of monads. Lean's actual definition is more complicated. One of the differences is that it distributes the definition over two type classes, called `monad` and `is_lawful_monad`.

6.5 The Option Monad

The following code extract shows how to register the type constructor `option : Type → Type` as a lawful monad:

```
namespace option

def pure {α : Type} : α → option α :=
option.some

def bind {α β : Type} : option α → (α → option β) → option β
| none     f := none
| (some a) f := f a

instance lawful_monad_option : lawful_monad option :=
{ pure       := @option.pure,
  bind       := @option.bind,
  pure_bind  :=
    begin
      intros α β a f,
      refl
    end,
  bind_pure  :=
    begin
      intros α m,
      cases m; refl
    end,
  bind_assoc :=
    begin
      intros α β γ f g m,
      cases m; refl
    end }

end option
```

The registration process requires us to provide five components: the `pure` and `bind` operations and the proofs of the three monadic laws. To avoid polluting the name space, we tidily put our code in an `option` namespace. For the first and second components of the record, we use the `@` syntax to make the type arguments explicit, in agreement with the format expected by `has_pure` and `has_bind`. The three proofs are straightforward.

6.6 The State Monad

The next monad is called the state monad. It provides an abstraction corresponding to a mutable state. For some programming languages, the compiler can detect the use of the state monad and translate state-monadic programs to more efficient imperative programs.

The type constructor is

```
def state (σ : Type) (α : Type) :=
σ → α × σ
```

For a given σ, we have that `state` σ `: Type → Type` is a monad. We abstract over the exact memory layout. We could use tuples or lists to represent memory, for example, and instantiate the abstract σ accordingly.

Intuitively, the $\sigma \to$ part of `state`'s definition provides the current state; the left component of the cartesian product, α, is where the result of a computation goes; and the

right component of the product, σ, can be used to provide a new state, overwriting the received one. Thus, the state is threaded through the monadic program.

We start by defining basic operations: two operations, `read` and `write`, to access the memory, and the monadic operations `bind` and `pure`:

```
namespace state

def read {σ : Type} : state σ σ
| s := (s, s)

def write {σ : Type} (s : σ) : state σ unit
| _ := ((), s)

def pure {σ α : Type} (a : α) : state σ α
| s := (a, s)

def bind {σ : Type} {α β : Type} (ma : state σ α)
  (f : α → state σ β) : state σ β
| s :=
  match ma s with
  | (a, s') := f a s'
  end
```

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 state `s` with `t` and returns an empty tuple.

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 `f` argument, yielding a result `a` and a new state `t`. These are passed to `g`, which returns a new result and a new state.

Registering the state monad as a lawful monad is similar as for `option`, but we need to reason about functional extensionality (`funext`). Namely, to prove `f = g`, it suffices to prove $\forall$`x, f x = g x`.

```
instance {σ : Type} : lawful_monad (state σ) :=
{ pure       := @state.pure σ,
  bind       := @state.bind σ,
  pure_bind  :=
    begin
      intros α β a f,
      apply funext,
      intro s,
      refl
    end,
  bind_pure  :=
    begin
      intros α m,
      apply funext,
      intro s,
      simp [bind],
      cases m s,
      refl
    end,
  bind_assoc :=
    begin
      intros α β γ f g m,
      apply funext,
```

```
      intro s,
      simp [bind],
      cases m s,
      refl
    end }

end state
```

As an example, the following state-monadic 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 in the monad.

```
def diff_list : list ℕ → state ℕ (list ℕ)
| []        := pure []
| (n :: ns) :=
  do
    prev ← read,
    if n < prev then
      diff_list ns
    else
      do
        write n,
        ns' ← diff_list ns,
        pure (n :: ns')
```

Notice how the state is accessed by `read` and `write`.

To execute the program, we must provide a start state. We also get back the last state, which is the largest element seen in the list or the start state:

```
#eval diff_list [1, 2, 3, 2] 0
#eval diff_list [1, 2, 3, 2, 4, 5, 2] 0
```

Output:

```
([1, 2, 3], 3)
([1, 2, 3, 4, 5], 5)
```

6.7 Example: Generic Iteration over a List

As a final example, we consider a generic monadic program `mmap` that iterates over a list and applies a function `f` to each element. The definition is recursive and monadic:

```
def mmap {m : Type → Type} [lawful_monad m] {α β : Type}
    (f : α → m β) : list α → m (list β)
| []       := pure []
| (a :: as) :=
  do
    b ← f a,
    bs ← mmap as,
    pure (b :: bs)
```

Notice that the function returns a single monad containing the resulting list, not a list of monads. Try to work out why the function is well typed and has the expected behavior.

The `mmap` function distributes over the append operator ++ of lists. The monadic notation is useful not only for defining functions but also for stating their properties:

```
lemma mmap_append {m : Type → Type} [lawful_monad m]
    {α β : Type} (f : α → m β) :
  ∀as as' : list α, mmap f (as ++ as') =
```

```
      do
        bs ← mmap f as,
        bs' ← mmap f as',
        pure (bs ++ bs')
| []         _    := by simp [mmap]
| (a :: as) as' := by simp [mmap, mmap_append as as']
```

Here is an example of how we could use `mmap`:

```
def nths {α : Type} (xss : list (list α)) (n : ℕ) :
  option (list α) :=
mmap (λxs, list.nth xs n) xss
```

The function `nths xss n` selects the (*n* + 1)st elements of each list in `xss`. It fails with `none` if one of the list is not long enough. For example, the call

```
#eval nths
  [[11, 12, 13, 14],
   [21, 22, 23],
   [31, 32, 33]] 2
```

produces the result `some [13, 23, 33]`.

6.8 Summary of New Syntax

Declaration

`class`	declares a structure as a type class

Monadic Notations

`do`	indicates the start of a monadic program
`>>=`	combines effectful computations

Chapter 7

Metaprogramming

Like most proof assistants, Lean can be extended with custom functionality, notably tactics. This kind of programming—programming the prover—is called metaprogramming. Lean's metaprogramming framework uses mostly the same notions and syntax as Lean's input language itself, so that we do not need to learn a different programming language.

These notes are partly inspired by an article by Gabriel Ebner et al. [5] and by Chapter 8 of *Programming in Lean* [2]. We strongly recommend that you read the chapter.

7.1 Overview

Lean's metaprogramming framework enables us to program the prover in its own language, using a monadic language. Abstract syntax trees, presented as inductive types in Lean, *reflect* internal data structures. For example, the type of expressions—i.e., terms—is viewed as an inductive type with ten constructors, but internally in Lean it is a C++ data structure. The prover's C++ internals are exposed through Lean interfaces, which we can use for accessing the current context and goal, unifying expressions, querying and modifying the environment, and setting attributes (e.g., `@[simp]`).

Most of Lean's predefined tactics are implemented as metaprograms, and not in C++. We can find these definitions by clicking on the name of a construct in Visual Studio Code while holding the control or command key.

Here are some example applications of metaprogramming:

- proof goal transformations (e.g., apply all safe introduction rules for connectives, put the goal in negation normal form);
- heuristic proof search (e.g., apply unsafe introduction rules for connectives and hypotheses with backtracking);
- decision procedures (e.g., for propositional logic, linear arithmetic);
- definition generators (e.g., Haskell-style `derive` for inductive types);
- advisor tools (e.g., lemma finders, counterexample generators);
- exporters (e.g., documentation generators);
- ad hoc proof automation (to avoid boilerplate or duplication).

A key benefit of Lean's metaprogramming framework is that users, who may not even be computer scientists, do not need to learn another programming language to write metaprograms. They can work with the same constructs and notation used to define ordinary objects in the prover's library. Everything in that library is available for metaprogramming purposes (e.g., $\mathbb{Z}$, `list`, algebraic structures). Metaprograms can be written and debugged in the same interactive environment, encouraging a style where formal libraries and supporting automation are developed at the same time.

The framework is articulated around three main types:

- the `tactic` monad, which contains the proof state, the environment, and more;
- the `name` type for storing structured names (e.g., `option.bind`);
- the `expr` type representing terms, types, proofs, and propositions as abstract syntax trees.

7.2 The Tactic Monad

The `tactic` monad combines the attributes of several kinds of monads. It is a state monad providing access among others to the list of goals, the environment (including all definitions and inductive types), notations, and attributes (e.g., the list of `@[simp]` rules).

Goals are represented as metavariables, standing for the missing proofs. Each metavariable has a type and a local context giving the constants and hypotheses. By the Curry–Howard correspondence, constants and hypotheses are not distinguished. Tactics have access to the elaborator, to elaborate expressions and compute their types.

The `tactic` monad also behaves like an option monad. The program `fail` *`msg`*, where *`msg`* is a text string, leaves the monad in an error state.

Moreover, the `tactic` monad is also an alternative monad. Namely, it provides an operator `<|>` for specifying alternatives. The program `s <|> t` first executes `s`; if `s` is a failure, then the state is reverted to its initial condition and `t` is executed instead.

Finally, `tactic` provides some tracing capabilities. We can use `trace msg`, where `msg` is a text string, or we can display the current proof state using `trace_state`.

Let us put the `tactic` monad to some use:

```
example :
  true :=
by do
  tactic.trace "Hello, Metacosmos!",
  tactic.triv
```

This first example is very modest: It uses the monad's tracing facilities to print the message "Hello, Metacosmos!" before applying the `triv` tactic, which can be used to prove trivial goals such as `true`. The `do` keyword signals entry into the `tactic` monad. The `by` keyword is necessary to enter tactic mode.

As is usually the case in computer science, we can introduce a layer of indirection. We can package our combination of tracing and proving as its own tactic, and gloriously apply it to prove `true`:

```
meta def hello_world : tactic unit :=
do
  tactic.trace "Hello, Metacosmos!",
  tactic.triv

example :
  true :=
by hello_world
```

The `hello_world` tactic's type is `tactic unit`. In general, tactics will have the type `tactic` σ, where σ is the type of some value returned by the tactic. Often our tactics will only have side effects on the state but return no value. In such cases, we return the empty tuple, of type `unit`. This is what the `triv` tactic returns, and hence this is what we return.

Also notice the presence of the `meta` keyword in front of the function definition. Any executable (i.e., not defined using the `noncomputable` keyword) Lean definition can be used as a metaprogram. In addition, we can put `meta` in front of a definition to indicate that is a *metadefinition*; these need not terminate but cannot be used in non-`meta` contexts. Metaprograms (whether defined with `meta` or not) can communicate with Lean

through *metaconstants*, which are implemented in C++ and have no logical meaning—i.e., they are opaque names. Metaprograms can also call other metaprograms.

Most tactics are designed to work on a proof state, but the Lean concept of a tactic is more general, and they can be used also on the top level, without a proof context, by using the `run_cmd` command:

```
run_cmd tactic.trace "Hello, Metacosmos!"
```

Tactics usually need to adapt to the situation where they are called. This means they must inspect the current goal. The following example prints the local context, the goal conclusions, and the current goal or "target":

```
example {α : Type} (a : α) :
  true :=
by do
  tactic.trace "local context:",
  tactic.local_context >>= tactic.trace,
  tactic.trace "goals:",
  tactic.get_goals >>= tactic.trace,
  tactic.trace "target:",
  tactic.target >>= tactic.trace,
  tactic.triv
```

The output is as follows:

- local context: `[α, a]`;
- goals: `[?m_1]`;
- target: `true`.

The goals are a list of variables of certain types that must be instantiated. They are normally not displayed in the proof state; instead, their types are shown. In the example, we have `?m_1 : true`.

The next example performs something somewhat more useful. It implements a `find_assumption` tactic that, like the predefined `assumption` tactic, looks for a hypothesis of the right type (i.e., the right proposition) and applies it to solve the current goal:

```
meta def exact_list : list expr → tactic unit
| []        := tactic.fail "no matching expression found"
| (e :: es) :=
  (do
     tactic.trace "trying ",
     tactic.trace e,
     tactic.exact e)
  <|> exact_list es

meta def find_assumption : tactic unit := do
  es ← tactic.local_context,
  exact_list es
```

The main function extracts all available hypotheses forming the local context, which are stored as expressions of type `expr`, and iterates through them. For each hypothesis, we first try to apply the predefined `exact` tactic (which exists also as a monadic program, like many other tactics). If this fails, the `<|>` operator passes control to the right branch, in which we continue recursively with the remaining hypotheses. If no hypotheses are left, we invoke `fail`.

A simple invocation follows:

```
example {p : Prop} {α : Type} (a : α) (h : p) :
  p :=
by do find_assumption
```

The tracing information informs us that `p`, α, `a`, and `h` have been tried in turn until the matching `h` was found.

7.3 Names and Expressions

Terms, including proof terms, are represented by an inductive type `expr` of "expressions" in the metaprogramming framework:

```
meta inductive expr : Type
| var      {} : nat → expr
| sort     {} : level → expr
| const    {} : name → list level → expr
| mvar        : name → name → expr → expr
| local_const : name → name → binder_info → expr → expr
| app         : expr → expr → expr
| lam         : name → binder_info → expr → expr → expr
| pi          : name → binder_info → expr → expr → expr
| elet        : name → expr → expr → expr → expr
| macro       : macro_def → list expr → expr
```

Briefly:

- `var` represents a bound variable, using a notation known as De Bruijn indices.
- `sort` is used to represent the type of types `Type`, `Type 1`, etc.
- `const` represents a constant, such as `list.reverse` or $\mathbb{N}$.
- `mvar` represents a metavariable, i.e., a variable with a question mark (e.g., `?m_1`).
- `local_const` represents a local constant in the proof state (e.g., `h`).
- `app` represents the application of a function to an argument.
- `lam` represents a λ binder.
- `pi` represents a Π binder.

Fortunately, we can often use higher-level interfaces to manipulate expressions. A lightweight way to construct an expression is using Lean's quoting mechanism, based on backticks. Expressions are enclosed in parentheses and preceded by one, two, or three backticks (`‘`).

An expression preceded by a single backtick—`‘(e)`—must be fully elaborated. It may not contain holes, as in the second example below:

```
run_cmd do
  let e : expr := ‘(list.map (λn : ℕ, n + 1) [1, 2, 3]),
  tactic.trace e

run_cmd do
  let e : expr := ‘(list.map _ [1, 2, 3]),
    -- fails (holes are disallowed)
  tactic.trace e
```

With two backticks—`‘‘(e)`—we have *pre-expressions*. These are Lean expressions that may contain some holes to be filled in later, based on some context:

```
run_cmd do
  let e₁ : pexpr := ‘‘(list.map (λn, n + 1) [1, 2, 3]),
  let e₂ : pexpr := ‘‘(list.map _ [1, 2, 3]),
  tactic.trace e₁,
  tactic.trace e₂
```

With three backticks—```(e)—we have pre-expressions without name checking. We can refer to `some_silly_name` even if it does not exist at the point when the pre-expression is parsed:

```
run_cmd do
  let e := ```(some_silly_name),
  tactic.trace e
```

A similar quoting mechanism exists for names. The syntax consists of one or two backticks, without parentheses:

```
run_cmd tactic.trace `some.silly.name
run_cmd tactic.trace ``true
run_cmd tactic.trace ``some.silly.name  -- fails (not found)
```

The single-backtick variant—`n—does not check whether the name exists; the two-backtick variant—``n—performs this check. Whether the check is desirable depends on the situation.

Sometimes we would like to plug in existing expressions into a larger expression. This can be achieved using *antiquotations*. These are announced by a prefix %% followed by a name from the current context. Antiquotations are available with one, two, and three backticks:

```
run_cmd do
  let x : expr := `(2 : ℕ),
  let e : expr := `(%%x + 1),
  tactic.trace e

run_cmd do
  let x : expr := `(@id ℕ),
  let e := ``(list.map %%x),
  tactic.trace e

run_cmd do
  let x : expr := `(@id ℕ),
  let e := ```(a _ %%x),
  tactic.trace e
```

Antiquotations can also be used to perform pattern matching:

```
example :
  1 + 2 = 3 :=
by do
  `(%%a + %%b = %%c) ← tactic.target,
  tactic.trace a,
  tactic.trace b,
  tactic.trace c,
  `(@eq %%α %%l %%r) ← tactic.target,
  tactic.trace α,
  tactic.trace l,
  tactic.trace r,
  tactic.exact `(refl _ : 3 = 3)
```

In the example, the two ← lines, corresponding to monadic bind operations, succeed only if the right-hand side has the right shape (which it has). Given that `target` is `1 + 2 = 3`, executing the first line binds the variables `a`, `b`, `r` to `1`, `2`, `3`, respectively. The second pattern looks at `1 + 2 = 3` through different lenses—namely, without syntactic sugar, as an expression of the form `(@eq` α `l r)`, for some α, `l`, `r`. This also succeeds.

7.4 Example: A Destructive Tactic

Next, we review two examples of metaprograms that accomplish a well-defined task. The first example is a tactic called `destruct_and` that automates the elimination of conjunctions in premises. Our aim is to automate proofs such as the following:

```
example {a b c d : Prop} (h : a ∧ (b ∧ c) ∧ d) :
  a :=
and.elim_left h

example {a b c d : Prop} (h : a ∧ (b ∧ c) ∧ d) :
  b :=
and.elim_left (and.elim_left (and.elim_right h))

example {a b c d : Prop} (h : a ∧ (b ∧ c) ∧ d) :
  b ∧ c :=
and.elim_left (and.elim_right h)
```

We would like in each case to simply write `destruct_and h`.

Our tactic relies on a helper function, which takes as argument the hypothesis `h` to use and its type—i.e., its proposition:

```
meta def destruct_and_helper : expr → expr → tactic unit
| `(%%a ∧ %%b) h :=
    tactic.exact h
    <|>
    (do
       ha ← tactic.to_expr ``(and.elim_left %%h),
       destruct_and_helper a ha)
    <|>
    (do
       hb ← tactic.to_expr ``(and.elim_right %%h),
       destruct_and_helper b hb)
| _            h := tactic.exact h
```

We perform pattern matching with antiquotations to see if the hypothesis `h` is of the form `a ∧ b`. If so, we try three things in turn. First, we try to invoke the `exact` tactic. If this fails, we try the left elimination rule, producing `and.elim_left h : a`, and continue recursively with that. The `to_expr` call converts a pre-expression into an expression, elaborating it. If our attempt at left elimination fails, we try right elimination instead.

The main function has very little to do:

```
meta def destruct_and (nam : name) : tactic unit :=
do
  h ← tactic.get_local nam,
  t ← tactic.infer_type h,
  destruct_and_helper t h
```

It retrieves the hypothesis `h` with the specified name `n`, extracts its type (i.e., its proposition), and calls the helper function.

We can now turn to our motivating examples and prove them again, this time using our new widget:

```
example {a b c : Prop} (h : a ∧ b ∧ c) :
  a :=
by destruct_and `h

example {a b c : Prop} (h : a ∧ b ∧ c) :
  c :=
```

```
by destruct_and ‘h

example {a b c : Prop} (h : a ∧ b ∧ c) :
  b ∧ c :=
by destruct_and ‘h
```

Unfortunately, we need to quote the name of the hypothesis with a backtick. We would need more (not particularly pleasant to write) code to provide proper parsing for our tactic's argument.

7.5 Example: A Solvability Advisor

Our second example enriches Lean with functionality that is available in the Isabelle/HOL proof assistant. It sometimes happens that a user states a lemma, proves it, and later realizes that the lemma already existed in the library. To help prevent this, we implement a `solve_direct` tactic, which goes through all lemmas in the database and checks whether one of them can be used to fully solve the current goal.

We will review the code in steps. The first step is a function `is_theorem` that returns true (`tt`) if a declaration is a theorem or an axiom and false (`ff`) otherwise:

```
meta def is_theorem : declaration → bool
| (declaration.defn _ _ _ _ _ _) := ff
| (declaration.thm _ _ _ _)      := tt
| (declaration.cnst _ _ _ _)     := ff
| (declaration.ax _ _ _)         := tt
```

We will use this function to filter out the declarations that do not interest us.

The next function returns the list of all theorem (and axiom) names:

```
meta def get_all_theorems : tactic (list name) :=
do
  env ← tactic.get_env,
  pure (environment.fold env [] (λdecl nams,
    if is_theorem decl then declaration.to_name decl :: nams
    else nams))
```

The `get_env` function (in fact, tactic) returns an object representing the environment. We can traverse it using `environment.fold`, which is applied on each declaration `decl` in turn and on an accumulator `thms`. We extend the accumulator with each declaration's name that satisfies the `is_theorem` predicate. At the end, `fold` returns the accumulator.

The `fold` function is a common idiom in functional programming. If you are not familiar with the concept, we recommend Chapter 6 of *Learn You a Haskell for Great Good!*[1]

Another component we need is a tactic that attempts to (fully) solve the goal using a theorem called `n`.

```
meta def solve_with_name (nam : name) : tactic unit :=
do
  cst ← tactic.mk_const nam,
  tactic.apply cst
    ({ md := tactic.transparency.reducible, unify := ff }
     : tactic.apply_cfg),
  tactic.all_goals tactic.assumption
```

`mk_const n` is used to produce an `expr` representing the proof from a name `n` (the theorem name).

[1] http://learnyouahaskell.com/higher-order-functions

The `apply` tactic applies an `expr` to the current goal. For performance, we pass an optional configuration, `{ md := transparency.reducible, unify := ff }`, which tells `apply` to perform less computational unfolding.

The `all_goals` tactical, in combination with `assumption`, is used to ensure that the hypothesis from the local context are used to fill in any remaining subgoals.

Finally, our `solve_direct` tactic goes through all theorems found by `get_all_theorems` and succeed with the first theorem that solves the current goal:

```
meta def solve_direct : tactic unit :=
do
  nams ← get_all_theorems,
  list.mfirst (λnam,
    do
      solve_with_name nam,
      tactic.trace ("directly solved by " ++ to_string nam))
    nams
```

The `list.mfirst` function applies a tactic to each element of a list until one application succeeds. We use `trace` to output the successful theorem to the user, so that they can apply the theorem directly instead of relying on `solve_direct`.

As a small refinement of the above, we propose a version of `solve_direct` that also looks for equalities stated in symmetric form—for example, if the goal is `l = r` but the theorem is `r = l`:

```
meta def solve_direct_symm : tactic unit :=
solve_direct
<|>
(do
   cst ← tactic.mk_const `eq.symm,
   tactic.apply cst,
   solve_direct)
```

Our solution is quite simple: We first try `solve_direct`, and if that fails, we apply symmetry of equality (the property `?l = ?r → ?r = ?l`) to change the goal and try `solve_direct` again.

7.6 Summary of New Syntax

Declaration

`meta`	prefixes declarations of metaprograms and metaconstants

Command

`run_cmd`	executes a tactic on the top level, without a proof context

Quotations

`` `n ``	quotes literal name, with checking
`` ``n ``	quotes literal name, without checking
`` `(e) ``	quotes fully elaborated expressions without holes
`` ``(e) ``	quotes pre-expressions, with name checking
`` ```(e) ``	quotes pre-expressions, without name checking
`%%x`	embeds an expression in a quotation

Tactic Monad Functions

`all_goals`	applies the given tactic to all goals
`get_env`	gives access to the list of all declarations in the environment
`get_goals`	gives the list of all proof holes to fill in
`get_local`	finds a local constant by name
`infer_type`	computes the type of an expression
`local_context`	gives the list of all local constants
`mk_const`	builds a constant from a name
`target`	gives the current goal to prove
`to_expr`	elaborates a pre-expression
`trace`	outputs a debug string
`trace_state`	outputs the current proof state

Part III

Program Semantics

Chapter 8

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 and to reason about the semantics.

This chapter is inspired by Chapter 7 of *Concrete Semantics: With Isabelle/HOL* [17], whose reading we recommend.

8.1 Motivation

A formal semantics helps specify and reason about the programming language itself, and about individual programs. It can form the basis of verified compilers, interpreters, verifiers, static analyses, type systems, etc. Without formal proofs, semantics, compilers, interpreters, verifiers, static analyses, type systems, etc., are almost always wrong.

Proof assistants are widely used in the area of programming language research. For example, every year, about 10%–20% of papers presented at POPL (Principles of Programming Languages) are partially or totally formalized. This is possible because comparatively little machinery (background libraries, tactics) is needed to get started, beyond inductive types and predicates and recursive functions, which are provided by the provers. The proofs tend to have lots of cases, which is a good match for computers. Proof assistants are convenient to keep track of what needs to be changed as we extend a programming language with more features.

8.2 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` p	(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 1.2 and similarly for Boolean expressions.

[1] Fans of backronyms might enjoy this one: Weak Hypothetical Imperative Language Example.

- We can decide that an arithmetic expression is simply a function from states to natural numbers (e.g., `state → ℕ`) and a Boolean expression is simply a predicate over states (e.g., `state → Prop` or `state → bool`). A *state* is a mapping from program variables to values. For example, the arithmetic expression `x + y + 1` would be represented by the function `λs : state, s "x" + s "y" + 1`, and the Boolean expression `a ≠ b` by the predicate `λs : state, s "a" ≠ s "b"`.

This corresponds 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. 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 explicitly about a program's syntax. A shallow embedding is more leightweight, because we can use it directly, without having to define a semantics (e.g., an `eval` function for `aexp`).

We will use a deep embedding of programs (which we find interesting) and a shallow embeddings of arithmetic and Boolean expressions (which we find boring). Our Lean definition of programs follows:

```
inductive program : Type
| skip {} : program
| assign  : string → (state → ℕ) → program
| seq     : program → program → program
| ite     : (state → Prop) → program → program → program
| while   : (state → Prop) → program → program

infixr ‘ ;; ‘:90 := program.seq
```

The infix operation `S ;; T` is provided as an alternative to `seq S T`.

The correspondence between the inductive type's constructors and the 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 (somewhat unusually) all possible variable names exist in the state and are assigned a value.

8.3 Big-Step Semantics

An *operational semantics* corresponds to an idealized, abstract 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.

The traditional (and also the most readable) way to specify such a semantics is through a formal system of derivation rules, in the style of the typing rules presented in Sections 1.1 and 3.4.

In a *big-step* (*operational*) *semantics* (also called *natural semantics*), judgments have the form `(S, s) ⟹ t` and the following intuitive interpretation:

> Starting in a state `s`, executing `S` terminates in the state `t`.

In keeping with the definition of WHILE programs, a state `s` is a function of type `string → ℕ`. An example of a judgment follows:

$$(\texttt{x := x + y; y := 0},\ [\texttt{x} \mapsto 3, \texttt{y} \mapsto 5]) \implies [\texttt{x} \mapsto 8, \texttt{y} \mapsto 0]$$

We use the informal notation `[x ↦ 3, y ↦ 5]` to denote the function `λvar, if var = "x" then 3 else if var = "y" then 5 else 0` and similarly for `[x ↦ 8, y ↦ 0]`. Intuitively, the judgment should hold.

The derivation rules for big-step semantics judgments are given below. The rules can be seen as an abstract interpreter for WHILE program.

$$\frac{}{(\texttt{skip},\ s) \Longrightarrow s}\ \textsc{Skip}$$

$$\frac{}{(x := a,\ s) \Longrightarrow s\{x \mapsto a\ s\}}\ \textsc{Asn}$$

$$\frac{(S,\ s) \Longrightarrow t \qquad (T,\ t) \Longrightarrow u}{(S;\ T,\ s) \Longrightarrow u}\ \textsc{Seq}$$

$$\frac{(S_1,\ s) \Longrightarrow t}{(\texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2,\ s) \Longrightarrow t}\ \textsc{If-True}\quad \text{if } \texttt{b s} \text{ is true}$$

$$\frac{(S_2,\ s) \Longrightarrow t}{(\texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2,\ s) \Longrightarrow t}\ \textsc{If-False}\quad \text{if } \texttt{b s} \text{ is false}$$

$$\frac{(S,\ s) \Longrightarrow t \qquad (\texttt{while}\ b\ \texttt{do}\ S,\ t) \Longrightarrow u}{(\texttt{while}\ b\ \texttt{do}\ S,\ s) \Longrightarrow u}\ \textsc{While-True}\quad \text{if } \texttt{b s} \text{ is true}$$

$$\frac{}{(\texttt{while}\ b\ \texttt{do}\ S,\ s) \Longrightarrow s}\ \textsc{While-False}\quad \text{if } \texttt{b s} \text{ is false}$$

Above, `a s` denotes the value of arithmetic expression `a` in state `s`, and similarly for `b s`. Moreover, the syntax $s\{x \mapsto n\}$ represents the state that is identical to s except that it maps the variable x to n. Formally: `s{x ↦ n} = (λvar, if var = x then n else s var)`. Exercise to the reader: Write the derivation tree for the judgment `(x := x + y; y := 0, [x ↦ 3, y ↦ 5]) ⟹ [x ↦ 8, y ↦ 0]`.

The derivation rules can be read intuitively. Consider Seq as an example. The rule can be read as follows:

> 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 the loop condition b is true in state s. If (1) executing S in state s leads to state t and (2) executing the loop `while b do S` from state t leads to state u, then executing the loop `while b do S` in state s leads to state u.

Another way to think about While-True is in terms of sequential composition. If the loop condition is true, `while b do S` should be equivalent to S; `while b do S`. The two premises of While-True correspond to the two premises of the instance of the Seq rule for the statement S; `while b do S`.

In Lean, a big-step semantics judgment is represented by an inductive predicate whose introduction rules correspond to the derivation rules above:

```
inductive big_step : program × state → state → Prop
| skip {s} :
  big_step (skip, s) s
| assign {x a s} :
  big_step (assign x a, s) (s{x ↦ a s})
| seq {S T s t u} (h₁ : big_step (S, s) t)
    (h₂ : big_step (T, t) u) :
  big_step (S ;; T, s) u
```

```
| ite_true {b : state → Prop} {S₁ S₂ s t} (hcond : b s)
    (hbody : big_step (S₁, s) t) :
  big_step (ite b S₁ S₂, s) t
| ite_false {b : state → Prop} {S₁ S₂ s t} (hcond : ¬ b s)
    (hbody : big_step (S₂, s) t) :
  big_step (ite b S₁ S₂, s) t
| while_true {b : state → Prop} {S s t u} (hcond : b s)
    (hbody : big_step (S, s) t)
    (hrest : big_step (while b S, t) u) :
  big_step (while b S, s) u
| while_false {b : state → Prop} {S s} (hcond : ¬ b s) :
  big_step (while b S, s) s
```

Using an inductive predicate as opposed to a recursive function allows us to cope with nontermination (e.g., a diverging `while`) and (for languages richer than WHILE) nondeterminism. It also arguably provides a nicer syntax, closer to the judgment rule syntax that is traditionally used in the scientific literature.

We make ample use of implicit arguments, denoted by curly braces, for the variables corresponding to the derivation rule's metavariables.

8.4 Lemmas about 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 interestingly, we can prove properties of the programming language, such as equivalence proofs between program constructs and determinism.

We will take determinism as an example, stated thus: If `(S, s)` $\Longrightarrow$ `l` and `(S, s)` $\Longrightarrow$ `r`, then `l = r`. Formally:

```
lemma big_step_unique {S s l r} (hl : (S, s) ⟹ l)
     (hr : (S, s) ⟹ r) :
  l = r :=
```

The proof is by rule induction over `(S, s)` $\Longrightarrow$ `l`, generalizing over `r`. We give an informal proof sketch:

Case Skip: The only way to have `(skip, s)` $\Longrightarrow$ `l` or `(skip, s)` $\Longrightarrow$ `r` is if `l = r = s`.

Case Asn: Similar to Skip.

Case Seq: We have the hypotheses `(S, s)` $\Longrightarrow$ `t`, `(T, t)` $\Longrightarrow$ `l`, `(S, s)` $\Longrightarrow$ `t'`, and `(T, t')` $\Longrightarrow$ `r` and the induction hypotheses $\forall$`r : state, (S, s)` $\Longrightarrow$ `r` $\rightarrow$ `t = r` and $\forall$`r : state, (T, t)` $\Longrightarrow$ `r` $\rightarrow$ `l = r`. From the first induction hypothesis together with `(S, s)` $\Longrightarrow$ `t'`, we derive `t = t'`. From the second induction hypothesis together with `(T, t')` $\Longrightarrow$ `r`, we derive `l = r`, as desired.

Case If-True: The only way to be in this case is if the condition `b s` is true. As a result, `(if b then S₁ else S₂, s)` $\Longrightarrow$ `r` can only have been derived using If-True; hence, `(S₁, s)` $\Longrightarrow$ `r`. The induction hypothesis is $\forall$`r : state, (S₁, s)` $\Longrightarrow$ `r` $\rightarrow$ `l = r`. We can apply `(S₁, s)` $\Longrightarrow$ `r` to it to obtain the desired result `l = r`.

Case If-False: Similar to If-True.

Case While-True: Similar to Seq.

Case While-False: Similar to Skip.

When reasoning about an inductive predicate, it is often convenient to use inversion rules (Section 5.3). Accordingly, we define the following rules:

```
@[simp] lemma big_step_skip_iff {s t} :
  (skip, s) ⟹ t ↔ t = s :=

@[simp] lemma big_step_assign_iff {x a s t} :
  (assign x a, s) ⟹ t ↔ t = s{x ↦ a s} :=

@[simp] lemma big_step_seq_iff {S₁ S₂ s t} :
  (S₁ ;; S₂, s) ⟹ t ↔ (∃u, (S₁, s) ⟹ u ∧ (S₂, u) ⟹ t) :=

@[simp] lemma big_step_ite_iff {b S₁ S₂ s t} :
  (ite b S₁ S₂, s) ⟹ t ↔
  (b s ∧ (S₁, s) ⟹ t) ∨ (¬ b s ∧ (S₂, s) ⟹ t) :=

lemma big_step_while_iff {b S s u} :
  (while b S, s) ⟹ u ↔
  (∃t, b s ∧ (S, s) ⟹ t ∧ (while b S, t) ⟹ u)
  ∨ (¬ b s ∧ u = s) :=

@[simp] lemma big_step_while_true_iff {b : state → Prop} {S s u}
    (hcond : b s) :
  (while b S, s) ⟹ u ↔
  (∃t, (S, s) ⟹ t ∧ (while b S, t) ⟹ u) :=

@[simp] lemma big_step_while_false_iff {b : state → Prop}
    {S s t} (hcond : ¬ b s) :
  (while b S, s) ⟹ t ↔ t = s :=
```

We add most of them to the simp set. We leave `big_step_while_true_iff` out because it could make the simplifier loop due to the presence on the right-hand side of a term that matches the left-hand side.

8.5 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 and the final state. This is for example too coarse-grained to reason about concurrency with an interleaving semantics. 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* give us a finer-coarsed view on computation. The transition relation has type `program × state → program × state → Prop`. Notice the presence of `program` in the type of the second operand. The intuition is that `(S, s) ⇒ (T, t)` indicates that executing one step of program `S` in state `s` leaves the program `T` to be executed, in state `t`. If there is no program left to be executed, we put `skip`.

An *execution* is a finite or infinite chain $(S_0, s_0) \Rightarrow (S_1, s_1) \Rightarrow \cdots$. A pair `(S, s)` is called a *configuration*; it is *final* if no transition of the form `(S, s) ⇒ _` is possible. Here is an example execution:

```
   (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])
```

The derivation rules for small-step semantics judgments are given below.

$$\frac{}{(x := a,\ s) \Rightarrow (\texttt{skip},\ s\{x \mapsto a\ s\})}\ \textsc{Asn}$$

$$\frac{(S,\ s) \Rightarrow (S',\ s')}{(S;\ T,\ s) \Rightarrow (S';\ T,\ s')}\ \textsc{Seq-Step}$$

$$\frac{}{(\texttt{skip};\ T,\ s) \Rightarrow (T,\ s)}\ \textsc{Seq-Skip}$$

$$\frac{}{(\texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2,\ s) \Rightarrow (S_1,\ s)}\ \textsc{If-True}\quad \text{if } \texttt{b s} \text{ is true}$$

$$\frac{}{(\texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2,\ s) \Rightarrow (S_2,\ s)}\ \textsc{If-False}\quad \text{if } \texttt{b s} \text{ is false}$$

$$\frac{}{(\texttt{while}\ b\ \texttt{do}\ S,\ s) \Rightarrow (\texttt{if}\ b\ \texttt{then (while}\ b\ \texttt{do}\ S,\ s\texttt{) else skip})}\ \textsc{While}$$

Unlike with the big-step semantics, there is no rule for the `skip` statement in the small-step semantics. This is because a state of the form `(skip, S)` is now considered final; `skip` is understood as the program whose execution is vacuous. By inspection of the rules, it is easy to see that a configuration is final if and only if its first component is `skip`.

There are two rules for 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 and, according to that, put the "then" or the "else" branch as the remaining computation to perform.

Finally, for the `while` loop, there is a single unconditional rule that unfolds 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 like this forever for infinite loops.

In Lean, the small-step semantics is specified as follows:

```
inductive small_step : program × state → program × state → Prop
| assign {x a s} :
  small_step (assign x a, s) (skip, s{x ↦ a s})
| seq_step {S T s s'} (S') :
  small_step (S, s) (S', s') →
  small_step (S ;; T, s) (S' ;; T, s')
| seq_skip {T s} :
  small_step (skip ;; T, s) (T, s)
| ite_true {b : state → Prop} {S₁ S₂ s} :
  b s → small_step (ite b S₁ S₂, s) (S₁, s)
| ite_false {b : state → Prop} {S₁ S₂ s} :
  ¬ b s → small_step (ite b S₁ S₂, s) (S₂, s)
| while {b : state → Prop} {S s} :
  small_step (while b S, s) (ite b (S ;; while b S) skip, s)
```

Equipped with a small-step semantics, we can *define* a big-step semantics as follows:

$$\texttt{(S, s)} \Longrightarrow \texttt{t} \text{ if and only if } \texttt{(S, s)} \Rightarrow\texttt{* (skip, t)}$$

where `r*` denotes the reflexive transitive closure of a relation `r`. Alternatively, if we have already defined a big-step semantics, we can *prove* the above equivalence theorem to *validate* our definitions.

We can prove that a configuration `(S, s)` is final if and only if `S = skip`; this ensures that we have not forgotten a derivation rule.

The main disadvantage of small-step semantics is that we now have two relations, $\Rightarrow$ and $\Rightarrow*$, and the derivation rules and proofs tend to be more complicated than with big steps.

8.6 Lemmas about the Small-Step Semantics

We can define inversion rules also about the small-step semantics. Here are three examples:

```
@[simp] lemma small_step_skip {S s t} :
  ¬ ((skip, s) ⇒ (S, t)) :=

lemma small_step_seq_iff {S T s Ut} :
  (S ;; T, s) ⇒ Ut ↔
  (∃S' t, (S, s) ⇒ (S', t) ∧ Ut = (S' ;; T, t))
  ∨ (S = skip ∧ Ut = (T, s)) :=

@[simp] lemma small_step_ite_iff {b S T s Us} :
  (ite b S T, s) ⇒ Us ↔
  (b s ∧ Us = (S, s)) ∨ (¬ b s ∧ Us = (T, s)) :=
```

A more important result is the connection between the big-step and the small-step semantics. If $\Rightarrow*$ denotes the reflexive transitive closure of the small-step relation $\Rightarrow$, we have the following theorem:

```
lemma big_step_iff_refl_trans_small_step {S s t} :
  (S, s) ⟹ t ↔ (S, s) ⇒* (skip, t) :=
```

Its proof is beyond the scope of this course. We refer to Chapter 7 of *Concrete Semantics: With Isabelle/HOL* [17] or to the Lean sources accompanying this text.

Chapter 9

Hoare Logic

Hoare logic is due to Tony Hoare. It is also called *axiomatic semantics*. If operational semantics corresponds to an abstract interpreter, Hoare logic corresponds to an abstract 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.

This chapter is inspired by Chapter 12 of *Concrete Semantics: With Isabelle/HOL* [17], whose reading we recommend.

9.1 Hoare Triples

Hoare logic is a framework for deriving valid correctness formulas in a mechanical way, using a set of axioms and proof rules. It allows us to reason directly on the program's syntax, without concerning ourselves with the operational semantics. The approach is mechanical in the sense that the applicability of an axiom or a proof rule can easily be checked using a computer or manually.

We start by introducing Hoare logic abstractly, without any connection to Lean; later we will see how we can embed Hoare logic judgments in Lean in a natural, useful way. 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 state variables. For the moment, we imagine the formulas as syntactic objects built using the familiar connectives, without being more specific. 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. 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 prevent normal termination.

Based on the intuitive interpretation of Hoare triples, all of the Hoare triples below are valid:

$$\texttt{\{true\}}\ \texttt{b := 4}\ \texttt{\{b = 4\}}$$

$$\texttt{\{a = 2\}}\ \texttt{b := 2 * a}\ \texttt{\{a = 2} \wedge \texttt{b = 4\}}$$

$$\texttt{\{b} \geq \texttt{0\}}\ \texttt{b := b + 1}\ \texttt{\{b} \geq \texttt{1\}}$$

$$\texttt{\{false\}}\ \texttt{skip}\ \texttt{\{b = 10\}}$$

$$\texttt{\{true\}}\ \texttt{while i < 10 do i := i + 1}\ \texttt{\{i = 10\}}$$

The first three Hoare triples should be fairly intuitive. 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. In a way, the triple is valid for the same reason that the assertion $\texttt{false} \to \texttt{b} = 10$ holds for any value of $\texttt{b}$. As for the fifth triple, there are no guarantees that control will escape the loop (e.g., if $\texttt{i} > 10$ initially), but if it does escape, then the loop's condition must be false and hence we have the postcondition $\texttt{i} = 10$.

9.2 Hoare Rules

The following is a complete set of derivation rules for reasoning about WHILE programs:

$$\frac{}{\{P\}\ \texttt{skip}\ \{P\}}\ \textsc{Skip}$$

$$\frac{}{\{Q[a/x]\}\ \texttt{x := a}\ \{Q\}}\ \textsc{Asn}$$

$$\frac{\{P\}\ S\ \{R\} \qquad \{R\}\ S'\ \{Q\}}{\{P\}\ \texttt{S; S'}\ \{Q\}}\ \textsc{Seq}$$

$$\frac{\{P \wedge b\}\ S\ \{Q\} \qquad \{P \wedge \neg b\}\ S'\ \{Q\}}{\{P\}\ \texttt{if b then S else S'}\ \{Q\}}\ \textsc{If}$$

$$\frac{\{I \wedge b\}\ S\ \{I\}}{\{I\}\ \texttt{while b do S}\ \{I \wedge \neg b\}}\ \textsc{While}$$

$$\frac{P' \to P \qquad \{P\}\ S\ \{Q\} \qquad Q \to Q'}{\{P'\}\ S\ \{Q'\}}\ \textsc{Conseq}$$

In the Asn rule, the syntax $\texttt{Q[a/x]}$ denotes the condition $\texttt{Q}$ with $\texttt{x}$ replaced by $\texttt{a}$. Here is another way to present the Asn rule:

$$\frac{}{\{Q[a]\}\ \texttt{x := a}\ \{Q[x]\}}\ \textsc{Asn}$$

where $\texttt{[x]}$ factors out *all* occurrences of $\texttt{x}$ in $\texttt{Q}$.

The Asn 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 by the following examples:

$$\{0 = 0\}\ \texttt{x := 0}\ \{x = 0\}$$
$$\{0 = 0 \wedge y = 5\}\ \texttt{x := 0}\ \{x = 0 \wedge y = 5\}$$
$$\{x + 1 \geq 5\}\ \texttt{x := x + 1}\ \{x \geq 5\}$$

Using elementary arithmetic, we can simplify the 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 $\texttt{R}$ that holds after executing $\texttt{S}$ and before executing $\texttt{S}'$. Here is an example of Seq in action:

$$\frac{\dfrac{}{\{a = 2\}\ \texttt{b := a}\ \{b = 2\}}\ \textsc{Asn} \qquad \dfrac{}{\{b = 2\}\ \texttt{c := b}\ \{c = 2\}}\ \textsc{Asn}}{\{a = 2\}\ \texttt{b := a; c := b}\ \{c = 2\}}\ \textsc{Seq}$$

In the While rule, the condition $\texttt{I}$ is called an *invariant*. It is pre- and postcondition of the loop itself but also of its body. The body's precondition is strengthened with the knowledge that $\texttt{b}$ must be true before executing the body. Moreover, the loop's postcondition is strengthened with the knowledge that $\texttt{b}$ must be false when the loop exits.

The Conseq rule is the only rule that has logical formulas among its premises, as opposed to only Hoare triples. These conditions must be discharged somehow—e.g., using pen and paper or a proof assistant. Conseq can be used to strengthen a precondition (i.e., make it more strict), weaken a postcondition (i.e. make it less strict), or both. An example derivation follows:

$$\dfrac{\texttt{x > 8} \to \texttt{x > 4} \qquad \dfrac{}{\texttt{\{x > 4\} y := x \{y > 4\}}}\,\textsc{Asn} \qquad \texttt{y > 4} \to \texttt{y > 0}}{\texttt{\{x > 8\} y := x \{y > 0\}}}\,\textsc{Conseq}$$

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; that is, we know which rule to apply in each case, simply by inspecting the program at hand. If the program is, say, an assignment, we apply the Asn rule—or Conseq.

Some of the above rules are nonlinear—i.e., their conclusions are not of the form `{P} ... {Q}` for distinct variables `P`, `Q`. This makes them inconvenient to apply. We combine some of the previous rules with Conseq to derive *linear* rules:

$$\dfrac{\texttt{P} \to \texttt{Q}}{\texttt{\{P\} skip \{Q\}}}\,\textsc{Skip}'$$

$$\dfrac{\texttt{P} \to \texttt{Q[a/x]}}{\texttt{\{P\} x := a \{Q\}}}\,\textsc{Asn}'$$

$$\dfrac{\texttt{\{P} \wedge \texttt{b\}\ S\ \{P\}} \qquad \texttt{P} \wedge \neg\texttt{b} \to \texttt{Q}}{\texttt{\{P\} while b do S \{Q\}}}\,\textsc{While}'$$

As an exercise, you could try to derive each of these rules from Skip, Asn, or While in conjunction with Conseq. This is not difficult.

9.3 A Semantic Approach to Hoare Logic

A natural way to encode Hoare logic in Lean would be to proceed roughly as we have done for the big- and small-step semantics: Define a syntactic notion of Hoare triple and an inductive predicate with one constructor for each of the core Hoare rule. To represent pre- and postconditions, use predicates on states (`state → Prop`). Then we could prove *soundness* with respect to, say, the big-step semantics, meaning the following. Assume `{P} S {Q}` is derivable. If `P s` and `(S, s) ⟹ t`, then `Q t`. This is merely a logical rendition of the informal meaning of a Hoare triple:

> If the *precondition* `P` is true before `S` is executed and the execution terminates normally, the *postcondition* `Q` is true at termination.

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 rules as lemmas, instead of stating them as introduction rules of an inductive predicate. In conjunction with the use of predicates to represent formulas, the approach is resolutely semantic.

Here is the definition of a Hoare triple (for partial correctness):

```
def partial_hoare (P : state → Prop) (S : program)
  (Q : state → Prop) : Prop :=
∀s t, P s → (S, s) ⟹ t → Q t
```

Instead of writing `partial_hoare 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:

```
lemma skip_intro :
  {* P *} skip {* P *} :=

lemma assign_intro (P : state → Prop) :
  {* λs, P (s{x ↦ a s}) *} assign x a {* P *} :=

lemma seq_intro (h₁ : {* P₁ *} S₁ {* P₂ *})
    (h₂ : {* P₂ *} S₂ {* P₃ *}) :
  {* P₁ *} S₁ ;; S₂ {* P₃ *} :=

lemma ite_intro (h₁ : {* λs, P s ∧ b s *} S₁ {* Q *})
    (h₂ : {* λs, P s ∧ ¬ b s *} S₂ {* Q *}) :
  {* P *} ite b S₁ S₂ {* Q *} :=

lemma while_intro (P : state → Prop)
    (h₁ : {* λs, P s ∧ b s *} S {* P *}) :
  {* P *} while b S {* λs, P s ∧ ¬ b s *} :=

lemma consequence (h : {* P *} S {* Q *}) (hp : ∀s, P' s → P s)
    (hq : ∀s, Q s → Q' s) :
  {* P' *} S {* Q' *} :=
```

They all have proofs in terms of the big-step semantics. Some of the triples—for example, the precondition in `assign_intro`—need to refer to the state explicitly. In such case, we can use a λ to access it. Recall that `P = (λs, P s)` by η-conversion. Moreover, for premise written informally as `P → Q`, in Lean we must write `∀s, P s → Q s`. This reflects the difference between syntactic formulas and semantic predicates.

The syntax `s{n ↦ a s}` in the assignment rule denotes the state that is identical to `s` at all points except `n` and that maps `n` to `a s`. This state update syntax and its automation are provided by the LoVelib library.

You might wonder, when looking at a rule such as `skip`, why its variable `P` is not declared. To lighten notation, we have declared most of the variables we need at the beginning of the Lean file:

```
variables {b : state → Prop} {a : state → ℕ} {x : string}
variables {S S₀ S₁ S₂ : program} {s s₀ s₁ s₂ t u : state}
variables {P P' P₁ P₂ P₃ Q Q' : state → Prop}
```

Lean then adds them as implicit arguments as necessary in the definitions and lemma statements that follow.

The formalization provide some more convenience rules, which can be derived from the core rules:

```
lemma consequence_left (P' : state → Prop)
    (h : {* P *} S {* Q *}) (hp : ∀s, P' s → P s) :
  {* P' *} S {* Q *} :=

lemma consequence_right (Q : state → Prop)
    (h : {* P *} S {* Q *}) (hq : ∀s, Q s → Q' s) :
  {* P *} S {* Q' *} :=

lemma skip_intro' (h : ∀s, P s → Q s):
  {* P *} skip {* Q *} :=
```

```
lemma assign_intro' (h : ∀s, P s → Q (s{x ↦ a s})):
  {* P *} assign x a {* Q *} :=

lemma seq_intro' (h₂ : {* P₂ *} S₂ {* P₃ *})
    (h₁ : {* P₁ *} S₁ {* P₂ *}) :
  {* P₁ *} S₁ ;; S₂ {* P₃ *} :=

lemma while_intro' (I : state → Prop)
    (h₁ : {* λs, I s ∧ b s *} S {* I *}) (hp : ∀s, P s → I s)
    (hq : ∀s, ¬ b s → I s → Q s) :
  {* P *} while b S {* Q *} :=
```

9.4 Example: Exchanging Two Variables

Let us use the Hoare logic to verify our first program: a program that exchanges the values of its variables `a` and `b`, using `t` for temporary storage. The program is defined as follows:

```
def SWAP : program :=
assign "t" (λs, s "a") ;;
assign "a" (λs, s "b") ;;
assign "b" (λs, s "t")
```

The proof is as follows:

```
lemma SWAP_correct (x y : ℕ) :
  {* λs, s "a" = x ∧ s "b" = y *} SWAP
  {* λs, s "a" = y ∧ s "b" = x *} :=
begin
  apply partial_hoare.seq_intro',
  apply partial_hoare.seq_intro',
  apply partial_hoare.assign_intro,
  apply partial_hoare.assign_intro,
  apply partial_hoare.assign_intro',
  simp { contextual := tt }
end
```

The applications of `seq_intro'` and `assign_intro` are syntax-directed: 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 ugly subgoal:

```
∀s : state,
  s "a" = x ∧ s "b" = y →
  s{"t" ↦ s "a"}{"a" ↦ s{"t" ↦ s "a"} "b"}{"b" ↦ s{"t" ↦ s "a"}
    {"a" ↦ s{"t" ↦ s "a"} "b"} "t"} "a" = y ∧
  s{"t" ↦ s "a"}{"a" ↦ s{"t" ↦ s "a"} "b"}{"b" ↦ s{"t" ↦ s "a"}
    {"a" ↦ s{"t" ↦ s "a"} "b"} "t"} "b" = x
```

Fortunately, `simp` can simplify the subgoal dramatically, and if we enable *contextual rewriting*—an option that tell `simp` to use the assumptions in the goal as rewrite rules—`simp` can even solve the subgoal.

9.5 Example: Adding Two Numbers

Our second example computes `m + n`, leaving the result in `n`, using only these primitive operations: `k + 1`, `k - 1`, and `k ≠ 0` (for arbitrary `k`):

```
def ADD : program :=
while (λs, s "n" ≠ 0)
  (assign "n" (λs, s "n" - 1) ;;
   assign "m" (λs, s "m" + 1))
```

The proof is more involved:

```
lemma ADD_correct (n₀ m₀ : ℕ) :
  {* λs, s "n" = n₀ ∧ s "m" = m₀ *} ADD
  {* λs, s "n" = 0 ∧ s "m" = n₀ + m₀ *} :=
begin
  refine partial_hoare.while_intro'
    (λs, s "n" + s "m" = n₀ + m₀) _ _ _,
  apply partial_hoare.seq_intro',
  apply partial_hoare.assign_intro,
  apply partial_hoare.assign_intro',
  { simp,
    -- this looks much better: 'simp' removed all updates
    intros s hnm hn0,
    rw ←hnm,
    -- subtracting on 'ℕ' is annoying
    cases s "n",
    { apply false.elim, apply hn0, refl },
    { simp [nat.succ_eq_add_one] } },
  { simp { contextual := tt } },
  { simp [not_not_iff] { contextual := tt } }
end
```

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 $n_0 + m_0$, where n_0 and m_0 correspond to the initial values of the program variables, as required by the precondition.

How did we come up with this invariant? Conceptually, most invariants have the form

work already done + work remaining to be done = desired result

where + in general denotes some suitable operator and not necessarily addition. In our example, the "work already done" is `m`, the "work remaining to be done" is `n`, and the "desired result" is $n_0 + m_0$. When exiting the loop, we have that $n = 0$, so the equation simplifies to "work already done = desired result." We can retrieve the desired result from `m`.

The application of `while_intro'` is performed using a tactic called `refine`, which works like `exact` but lets us specify holes to be filled later.

After applying the obvious rules on the loop body, we are left with three subgoals. The only nontrivial subgoal is the proof that executing the body maintains the invariant. There is some ugly arithmetic on natural numbers necessary due to the use of the minus operator. Our proof simply performs a case distinction to separate the zero case, where $- 1$ has no effect ($0 - 1 = 0$), and the nonzero case, where $- 1$ has an effect. Each case is easy to discharge using the simplifier with contextual rewriting.

9.6 A Verification Condition Generator

Verification condition generators (VCGs) are programs that apply Hoare logic rules automatically, producing *verification conditions* that must be proved by the user. The user must provide strong enough loop invariants, as annotations in their programs. Hundreds of program verification tools are based on these principles. Often these are based on an

extension called separation logic,[1] which is sometimes called "the Hoare logic of the 21st century."

VCGs typically work backwards from the postcondition, using backward rules—i.e., rules stated to have an arbitrary `Q` as their postcondition; this works well because the key rule ASN, for assignment, is backward.

We can use Lean's metaprogramming framework to define a simple VCG. First, we introduce a constant called `while_inv` that carries an invariant `I` in addition to the loop condition `b` and body `S`:

```
def program.while_inv (I : state → Prop) (c : state → Prop)
  (p : program) : program :=
while c p

export program (while_inv)
```

We provide two Hoare rules for the construct: a backward rule and a linear rule. Both are justified in terms of the `while_intro'` rule we presented earlier:

```
lemma while_inv_intro {I : state → Prop}
    (h₁ : {* λs, I s ∧ b s *} S {* I *})
    (hq : ∀s, ¬ b s → I s → Q s) :
  {* I *} while_inv I b S {* Q *} :=
while_intro' I h₁ (assume s hs, hs) hq

lemma while_inv_intro' {I : state → Prop}
    (h₁ : {* λs, I s ∧ b s *} S {* I *}) (hp : ∀s, P s → I s)
    (hq : ∀s, ¬ b s → I s → Q s) :
  {* P *} while_inv I b S {* Q *} :=
while_intro' I h₁ hp hq
```

The rules simply use the invariant annotation as their invariant. When using the framework, we will have to be careful to provide a suitable invariant, because there will not be a second chance to provide one.

The code of the VCG is fairly concise:

```
meta def is_meta : expr → bool
| (expr.mvar _ _ _) := tt
| _                 := ff

meta def vcg : tactic unit := do
  t ← tactic.target,
  match t with
  | `({* %%P *} %%S {* _ *}) :=
    match S with
    | `(program.skip)           :=
      tactic.applyc
        (if is_meta P then ``partial_hoare.skip_intro
         else ``partial_hoare.skip_intro')
    | `(program.assign _ _)     :=
      tactic.applyc
        (if is_meta P then ``partial_hoare.assign_intro
         else ``partial_hoare.assign_intro')
    | `(program.seq _ _)        :=
      tactic.applyc ``partial_hoare.seq_intro';
        vcg
    | `(program.ite _ _ _)      :=
```

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

```
      tactic.applyc ``partial_hoare.ite_intro;
        vcg
    | `(program.while_inv _ _ _) :=
      tactic.applyc
          (if is_meta P then ``partial_hoare.while_inv_intro
           else ``partial_hoare.while_inv_intro');
        vcg
    | _                          :=
      tactic.fail (to_fmt "cannot analyze " ++ to_fmt S)
    end
  | _                        := skip
  end
```

The `is_meta` helper function determines whether a Lean term is a metavariable (i.e., a `?` variable such as `?P`).

The VCG proper extracts the current goal—the target—and inspects it. If it is a Hoare triple, it inspects its precondition `P` and statement `S`. If the precondition is a metavariable, then it applies a backward rule if there exists one (e.g., the standard assignment rule `assign_intro`) because this will usefully instantiate the metavariable. Otherwise, a linear rule is used, with an arbitrary variable as its precondition, which can be matched against the goal's precondition. For `while` loops, we only consider programs that use `while_inv`, because we cannot guess the invariant programmatically.

The VCG calls itself recursively on all newly emerging goals (via the `;` tactic combinator) for the compound statements `seq`, `ite`, and `while_inv`.

9.7 Example: Adding Two Numbers, Revisited

Using the verification condition generator, we can revisit the proof of `ADD` presented above:

```
lemma ADD_correct₂ (n₀ m₀ : ℕ) :
  {* λs, s "n" = n₀ ∧ s "m" = m₀ *} ADD
  {* λs, s "n" = 0 ∧ s "m" = n₀ + m₀ *} :=
show
  {* λs, s "n" = n₀ ∧ s "m" = m₀ *}
  while_inv (λs, s "n" + s "m" = n₀ + m₀) (λs, s "n" ≠ 0)
    (assign "n" (λs, s "n" - 1) ;;
     assign "m" (λs, s "m" + 1))
  {* λs, s "n" = 0 ∧ s "m" = n₀ + m₀ *},
begin
  vcg;
    simp { contextual := tt },
  intros s hnm hn,
  rw ←hnm,
  cases s "n",
  { apply false.elim, apply hn, refl },
  { simp [nat.succ_eq_add_one] }
end
```

There are two main points to observe.

First, we use `show` to annotate the while loop with an invariant. Recall that the `show` command lets us restate the goal in a computationally equivalent way. Here, we use this facility to replace `while` by `while_inv`, whose definition is in terms of `while`. The program and its pre- and postconditions are otherwise the same as in the lemma statement.

Second, we invoke `vcg` as the first proof step. This will apply all the necessary Hoare rules and leave us with some proof goals. We first simplify them with contextual rewriting

enabled; this already takes care of two of the three goals. For the third goal, the proof is as before.

9.8 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 program `while true do skip`. This is admittedly too liberal: If we are asked to provide, say, a sorting algorithm, we should not simply give `while true do skip` as the answer.

Total correctness is a stronger notion asserts that the program terminates normally. One reason to focus on partial correctness is that it is a necessary component of total correctness. It is also simpler, which is a benefit in the classroom.

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, the following is an equivalent formulation:

> 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. The Hoare rule for `while` must now be annotated by a *variant* `V`—a natural number that decreases with each iteration:

$$\frac{\texttt{[I} \wedge \texttt{b} \wedge \texttt{V = n] S [I} \wedge \texttt{V < n]}}{\texttt{[I] while b do S [I} \wedge \neg\texttt{b]}}\ \textsc{While}$$

The variable `n` is a logical variable whose scope is the entire premise, whereas `V` is a metavariable standing for the variant. For the example above, we would take `10 - i` as the variant. A derivation tree for this example is given on Wikipedia.[2]

9.9 Summary of New Syntax

Tactics

`refine` applies a rule that may contain holes (_) left as subgoals

`simp ... { contextual := tt }` simplifies using assumptions in the goal as rewrite rules

[2] `https://en.wikipedia.org/wiki/Hoare_logic#While_rule_for_total_correctness`

Chapter 10

Denotational Semantics

We review a third way to specify the semantics of a programming language: denotational semantics. Denotational semantics attempts to directly specify the semantics of programs as a mathematical object. If operational semantics corresponds to an abstract interpreter and Hoare logic corresponds to an abstract verifier, then denotational semantics corresponds to an abstract compiler.

This chapter is inspired by Chapter 11 of *Concrete Semantics: With Isabelle/HOL* [17], whose reading is recommended.

10.1 Motivation

A *denotational semantics* defines the meaning of each program as a mathematical object and can be seen as a function

$$[\![\ ]\!] : \texttt{syntax} \to \texttt{semantics}$$

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. This compositionality makes it possible to reason equationally about programs. The compositionality requirement disqualifies the trivial definition

$$[\![\texttt{S}]\!] = \texttt{\{st | (S, st.1)} \Longrightarrow \texttt{st.2\}}$$

in terms of the big-step semantics $\Longrightarrow$ because that semantics is not compositional: The semantics of a while loop is given in terms of itself, not of its loop body.

In essence, we want structurally recursive equations of the form

$$\begin{aligned}
[\![\texttt{S ; S'}]\!] &= \ldots\ [\![\texttt{S}]\!]\ \ldots\ [\![\texttt{S'}]\!]\ \ldots \\
[\![\texttt{if b then S else S'}]\!] &= \ldots\ [\![\texttt{S}]\!]\ \ldots\ [\![\texttt{S'}]\!]\ \ldots \\
[\![\texttt{while b do S}]\!] &= \ldots\ [\![\texttt{S}]\!]\ \ldots
\end{aligned}$$

An evaluation function such as $\texttt{eval} : \texttt{aexp} \to ((\texttt{var} \to \mathbb{Z}) \to \mathbb{Z})$ on arithmetic expressions is a denotational semantics. Now we want the same for imperative programs. We need to develop some mathematical notions to reach the point where we can formulate the denotational semantics.

10.2 A Relational Denotational Semantics

Denotational semantics for deterministic languages are normally given as a *function* from prestate to poststate, but we find *relations* more convenient to manipulate. Hence we present a relational denotational semantics, following *Concrete Semantics* [17]. The relational approach was also used for the big-step semantics, which took the form of a

predicate of type `state → state → Prop`. In contrast, a denotational semantics of a program will be a mathematical object of type `set (state × state)`.

To a mathematician working informally, `state → state → Prop` and `set (state × state)` are "the same," since there is a one-to-one correspondence between predicates on pairs and subsets of the cartesian product. To a type theorist, `state → state → Prop` and `set (state × state)` are "the same" up to currying, since `set α = α → Prop` by definition. So why make this distinction? One reason is the different perspective afforded by the different structures. Another is for efficiency of notation or of computation. The final reason is that many systems of formal logic enforce a strict separation between predicates and their arguments, making it formally nonsensical to even ask whether `state → state → Prop` and `set (state × state)` are "the same."

We call our denotational semantics `den`, with ⟦ ⟧ as syntactic sugar. Let us first study the first four equations of the definition, keeping the `while` case for later:

```
def den : program → set (state × state)
| skip          := Id state
| (assign n a)  := {x | x.2 = x.1{n ↦ a x.1}}
| (seq S₁ S₂)   := den S₁ ○ den S₂
| (ite b S₁ S₂) := (den S₁ ⨾ b) ∪ (den S₂ ⨾ λs, ¬ 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. Recall that if `x` is a pair, then `x.1` and `x.2` give access to its first and second components.

The semantics of sequential composition is elegantly expressed as a relational composition ○, which is defined such that `r₁ ○ r₂ = {x | ∃y, (x.1, y) ∈ r₁ ∧ (y, x.2) ∈ r₂}`.

The semantics of `ite` is the union of the behaviors of the two branches, restricted to include only the tuples whose first component makes the Boolean condition true or false, according to the branch. The restriction operator is defined as `r ⨾ p = {x | p x.1 ∧ x ∈ r}`.

The difficulties arise when we try to define the `while` case. We would like to write

```
| (while b S)   :=
  ((den S ○ den (while b S)) ⨾ b) ∪ (Id state ⨾ λs, ¬ b s))
```

but this is not well founded due to the unmodified recursive call to `while b S`. So we need to write this differently. What we are looking for is a solution for `X` in the equation

```
X = ((den S ○ X) ⨾ b) ∪ (Id state ⨾ λs, ¬ b s))
```

In other words, we are looking for a fixpoint. The rest of this chapter is concerned with building a fixpoint operator `lfp` that will allow us to define the `while` case as well:

```
| (while b S)   :=
  complete_lattice.lfp
    (λX, ((den S ○ X) ⨾ b) ∪ (Id state ⨾ λs, ¬ b s))
```

The union and restriction operators are as for the `ite` case.

10.3 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` (e.g., `f := nat.succ`), or there may be many fixpoints (e.g., `f := λx, x`). Under some conditions on `f`, a unique *least fixpoint* and a unique *greatest fixpoint* are guaranteed to exist.

Consider the following fixpoint equation:

$$\texttt{X} = \lambda \texttt{n} : \mathbb{N},\ \texttt{n} = 0 \lor \exists \texttt{m} : \mathbb{N},\ \texttt{n} = \texttt{m} + 2 \land \texttt{X m}$$

This equation is the β-contracted version of an equation that has more obviously the right format, with $\texttt{X} : \mathbb{N} \to \texttt{Prop}$ and $\texttt{f} := (\lambda(\texttt{p} : \mathbb{N} \to \texttt{Prop})\ (\texttt{n} : \mathbb{N}),\ \texttt{n} = 0 \lor \exists \texttt{m} : \mathbb{N},\ \texttt{n} = \texttt{m} + 2 \land \texttt{p m})$:

$$\texttt{X} = (\lambda(\texttt{p} : \mathbb{N} \to \texttt{Prop})\ (\texttt{n} : \mathbb{N}),\ \texttt{n} = 0 \lor \exists \texttt{m} : \mathbb{N},\ \texttt{n} = \texttt{m} + 2 \land \texttt{p m})\ \texttt{X}$$

It turns out that this equation admits only one fixpoint. The fixpoint equation uniquely specifies X as the set of even natural numbers.

In general, the least and greatest fixpoint may be different. Consider the equation

$$\texttt{X} = (\lambda \texttt{x},\ \texttt{x})\ \texttt{X}$$

where $\texttt{X} : \mathbb{N} \to \texttt{Prop}$. The least fixpoint of this equation is $\lambda \_,\ \texttt{False}$ and the greatest fixpoint is $\lambda \_,\ \texttt{True}$. Conventionally, we have that False < True and thus $(\lambda \_,\ \texttt{False}) < (\lambda \_,\ \texttt{True})$. Similarly, $\emptyset <$ `@set.univ` α if α is inhabited (i.e., if it contains at least one element).

Incidentally, Lean's inductive predicates correspond to least fixpoints, but they are built into Lean's calculus of inductive constructions. We implicitly made use of this observation when we defined the operational semantics as an inductive data type.

For the semantics of programming languages:

- X will have type `set (state × state)`, representing relations between states;
- 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`? A program should be prohibited from looping infinitely: Each time a loop is evaluated, it must terminate after some number of iterations n. Let $\texttt{f}^n$ denote the n-fold composition of f, corresponding to performing at most n iterations, then we have $\texttt{X} = \texttt{f}^n(\emptyset)$ in this case. More generally, we always have

$$\texttt{X} = \texttt{f}^0(\emptyset) \cup \texttt{f}^1(\emptyset) \cup \texttt{f}^2(\emptyset) \cup \cdots$$

Kleene's fixpoint theorem[1] states that the least fixpoint `lfp f` will produce the value of X satisfying this equation. The greatest fixpoint would also consider cyclic and diverging executions.

10.4 Monotone Functions

Let α and β be arbitrary types, each equipped with a partial order $\leq$. We say that the function $\texttt{f} : \alpha \to \beta$ is *monotone* if $\texttt{a} \leq \texttt{b} \to \texttt{f a} \leq \texttt{f b}$ for all a, b.

Many operations on sets (e.g., $\cup$), relations (e.g., $\bigcirc$), and functions (e.g., const) are monotone. The set of monotone functions is also well behaved: The identity function is monotone, the composition of monotone functions is again monotone, and so on.

All monotone functions $\texttt{f} : \alpha \to \alpha$, where α is a "complete lattice" (introduced below), admit least and greatest fixpoints.

Here is an example of a nonmonotone function on sets:

$$\texttt{f s} = \begin{cases} \texttt{s} \cup \{\texttt{a}\} & \text{if a} \notin \texttt{s} \\ \emptyset & \text{otherwise} \end{cases}$$

Notice that $\emptyset \subseteq \{\texttt{a}\}$ but $\texttt{f}\ \emptyset = \{\texttt{a}\} \not\subseteq \emptyset = \texttt{f}\ \{\texttt{a}\}$.

[1] https://en.wikipedia.org/wiki/Kleene_fixed-point_theorem

10.5 Complete Lattices

To define the least fixpoint for sets, we only need one operation: intersection $\bigcap$. The same definition works in 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

- a partial order $\leq$ `:` $\alpha \to \alpha \to$ `Prop` (i.e., reflexive, transitive, and antisymmetric);
- an operator $\bigsqcap$ `:` `set` $\alpha \to \alpha$, called *infimum*.

The $\bigsqcap$`s` operator satisfies the following two conditions:

- $\bigsqcap$`s` is a lower bound of `s`: $\bigsqcap$`s` $\leq$ `b` for all `b` $\in$ `s`.
- $\bigsqcap$`s` is the greatest lower bound: `b` $\leq \bigsqcap$`s` for all `b` such that $\forall$`x` $\in$ `s`, `b` $\leq$ `x`.

We are justified in writing "*the* greatest lower bound": If α is a partial order, for any `s` `:` `set` α there is at most one `a` `:` α satisfying both conditions on the infimum.

Warning: $\bigsqcap$`s` is not necessarily an element of `s`. For example, an open interval]*a*, *b*[has infimum *a*, which is not in]*a*, *b*[.

Here are some examples of complete lattices:

- `enat` := $\mathbb{N} \cup$ {$-\infty$, ∞};
- `ereal` := $\mathbb{R} \cup$ {$-\infty$, ∞};
- `set` α is a complete lattice with respect to $\subseteq$ and $\bigcap$ for all types α;
- `Prop` is a complete lattice with respect to $\to$ and $\forall$, i.e., $\bigsqcap$`s` := $\forall$`p` $\in$ `s`, `p`.

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 *non*examples of complete lattices: $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{R}$. The issue is that there is no largest element, which the infimum for $\emptyset$ must be. Another nonexample: `erat` := $\mathbb{Q} \cup$ {$-\infty$, ∞}. The issue is that $\bigsqcap$`{q | 2 < q * q}` = `sqrt 2` is not in $\mathbb{Q}$.

Finally, we define

$$\texttt{lfp f} := \bigsqcap\texttt{\{x | f x} \leq \texttt{x\}}$$

The Knaster–Tarski theorem[2] 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` $\to$ `lfp f` $\leq$ `X`.

With `lfp`, we can complete the definition of the denotational semantics, as announced:

```
| (while b S)   :=
  complete_lattice.lfp
    (λX, ((den S ○ X) ⨅ b) ⊔ (Id state ⨅ λs, ¬ b s))
```

10.6 A Relational Denotational Semantics, Continued

The proof of the pudding is in the eating. We defined the semantics of `while` in terms of the `lfp` operator, but perhaps monotonicity—which guarantees the existence of a least fixpoint—is not met. To quell such doubts, we prove the lemma

```
lemma den_while (S : program) (b : state → Prop) :
  ⟦while b S⟧ = ⟦ite b (S ;; while b S) skip⟧ :=
begin
  apply complete_lattice.lfp_eq,
  apply monotone.union,
  { apply sorry_lemmas.monotone_restrict,
```

[2] https://en.wikipedia.org/wiki/Knaster-Tarski_theorem

```
    apply sorry_lemmas.monotone_comp,
    { exact monotone.const _ },
    { exact monotone.id } },
  { exact monotone.const _ }
end
```

The lemma gives us a convenient way to unfold one iteration of a loop. The first proof step expands the least fixpoint, exploiting definitional equality to recognize the semantics of an `ite` hiding in the semantics of a `while`. The rest of the proof is all about proving monotonicity of the body of the fixpoint operator.

Another important result is the connection between the denotational and the big-step semantics:

```
lemma den_eq_bigstep (S : program) (s t : state) :
  (s, t) ∈ ⟦S⟧ ↔ (S, s) ⟹ t :=
```

Its proof is beyond the scope of this course. We refer to Chapter 11 of *Concrete Semantics: With Isabelle/HOL* [17] or to the Lean files accompanying this book.

Part IV

Mathematics

Chapter 11

Logical Foundations of Mathematics

In this chapter, we dive deeper into the logical foundations of Lean.

11.1 Type Universes

In type theory—and in particular in Lean—not only all terms have a type, but also all types have a type themselves. We have already seen one occurrence of this principle. The Curry–Howard correspondence tells us to view proofs as terms and propositions as types. For example:

```
@and.intro : ∀a b, a → b → a ∧ b
```

Thus, a proposition such as `∀a b, a → b → a ∧ b` is a type, and this type has in turn a type, namely `Prop`:

```
∀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, namely, `Type`:

```
ℕ : Type
Prop : Type
```

What is the type of `Type`? It may seem like the simplest solution to assign `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` again cannot be assigned any of the other types introduced so far without causing contradictions, leading to the introduction of `Type 2`, and so on:

```
Type : Type 1
Type 1 : Type 2
Type 2 : Type 3
⋮
```

In fact, `Type` is an abbreviation for `Type 0`. If we want to incorporate `Prop` in this hierarchy, we can also use the syntax `Sort u`, where `Sort 0` is the same as `Prop` and `Sort (u + 1)` is the same as `Type u`.

All of these types containing other types are called (*type*) *universes*, and the `u` in the expression `Sort u` is a `universe level`. Although universe levels look like terms of type `ℕ`, they are in fact neither of type `ℕ` nor even terms.

To formalize most of computer science and mathematics, `Type` (i.e., `Type 0`) suffices. Instead of using `Type` everywhere though, you can make your theorems slightly more general without having to think about universe levels by writing `Type*`, which abbreviates

`Type _`. It leaves it up to Lean figure out the right universe level or to create a fresh universe variable.

11.2 The Peculiarities of Prop

Although `Prop` seems to fit nicely into the hierarchy of type universes, it is different from the other universes in many respects.

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 type 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:

$$\frac{\Gamma \vdash \alpha \,\texttt{: Type u} \qquad \Gamma\texttt{, (a : } \alpha\texttt{)} \vdash \beta\ \texttt{a : Type v}}{\Gamma \vdash \texttt{(}\Pi\texttt{a : } \alpha\texttt{, } \beta\ \texttt{a)} \,\texttt{: Type (max u v)}}\ \text{PI}$$

This behavior of the universes `Type v` is called *predicativity*.

However, it is convenient to have `Prop` behave differently. We would like $\forall$`(a : Prop) , a` $\to$ `a` to be of type `Prop`. Unfolding the syntactic sugar $\forall$ and $\to$, this expression is the same as Π`(a : Prop) (x : a), a`. If we had `Type u` instead of `Prop`, the above typing rule would yield

$$\texttt{(}\Pi\texttt{(a : Type u) (x : a), a) : Type (u + 1)}$$

because `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 $\forall$`(a : Prop ), a` $\to$ `a` (which equals Π`(a : Prop) (x : a), a`) to be of type `Prop` anyway, we have a special typing rule for Π expressions involving `Prop`:

$$\frac{\Gamma \vdash \alpha \,\texttt{: Sort u} \qquad \Gamma\texttt{, (a : } \alpha\texttt{)} \vdash \beta\ \texttt{a : Prop}}{\Gamma \vdash \texttt{(}\Pi\texttt{a : } \alpha\texttt{, } \beta\ \texttt{a)} \,\texttt{: Prop}}\ \text{PI}$$

This is called *predicativity* of `Prop`. The rule yields

$$\texttt{(}\Pi\texttt{(a : Prop) (x : a), a) : Prop}$$

as desired. The two typing rules for Π expressions above can be summarized in one rule

$$\frac{\Gamma \vdash \alpha \,\texttt{: Sort u} \qquad \Gamma\texttt{, a : } \alpha \vdash \beta\ \texttt{a : Sort v}}{\Gamma \vdash \texttt{(}\Pi\texttt{a : } \alpha\texttt{, } \beta\ \texttt{a)} \,\texttt{: Sort (imax u v)}}\ \text{PI}$$

where

```
imax u 0 = 0
imax u v = max u v   if v ≠ 0
```

In other systems, such as Agda, all type universes are predicative.

Proof Irrelevance

A second difference between `Prop` and `Type` is proof irrelevance. It means that any two proofs of the same proposition are equal:

```
lemma proof_irrel {a : Prop} (h₁ h₂ : a) :
  h₁ = h₂ :=
by refl
```

In Lean, this equality is even a definitional equality, allowing us to prove that any two proofs of the same proposition are equal using the `refl` 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.

Other systems and logics make different choices in this matter. For example, Agda and Coq are proof relevant by default, but are 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.

Small Elimination

A further difference between `Prop` and `Type` is that `Prop` does not allow *large elimination*, meaning that it is impossible to extract data from a proof of a proposition. Consider the following inductive type:

```
inductive square_roots (sq : ℤ) : Type u
| mk (z : ℤ) (h : sq = z * z) : square_roots
```

The type `square_roots sq` contains all square roots of a given integer `sq`. It contains no elements if `sq` is not square, one element if `sq = 0`, and two elements otherwise. For example:

```
def two_as_root : square_roots 4 :=
square_roots.mk 2 (by refl)

def neg_two_as_root : square_roots 4 :=
square_roots.mk (-2) (by refl)
```

A large eliminator can be defined as follows:

```
def large_elim {sq : ℤ} {α : Sort v} :
  square_roots sq → (Π(z : ℤ) (h : sq = z * z), α) → α
| (square_roots.mk z h) f := f z h
```

This eliminator's first explicit argument an element of `square_roots sq`. The second one is a function that converts a number and a proof that `sq` is the square of that number into an element of an arbitrary type α. Given those two things, it can produce the element of α corresponding to the given element of `square_roots sq`. For example, it can be used to extract the number that was used to construct the given element of `square_roots`:

```
#eval large_elim two_as_root (λz _, z)       -- result: 2
#eval large_elim neg_two_as_root (λz _, z)   -- result: -2
```

Therefore, the large eliminator can be used to prove that two elements of `square_roots` are different if they were constructed using different roots:

```
example : two_as_root ≠ neg_two_as_root :=
begin
  have hne : large_elim two_as_root (λz _, z)
             ≠ large_elim neg_two_as_root (λz _, z) :=
  begin
    intro h,
    cases h
  end,
  intro hr,
  apply hne,
  rw hr,
end
```

(This proof could be written more elegantly using the `cases` tactic.)

Moreover, the large eliminator can be used to show that all numbers with a square root are nonnegative:

```
example (sq : ℤ) (h : square_roots sq) : sq ≥ 0 :=
begin
  apply large_elim h,
  intros z hz,
  rw hz,
  apply mul_self_nonneg
end
```

(This proof could be written more elegantly using the `cases` tactic, too)

Now we make a second definition that strongly resembles `square_roots`. The only difference is that we define it as an inductive predicate, i.e., we put into the `Prop` universe instead of `Type u`.

```
inductive square_number (sq : ℤ) : Prop
| mk (z : ℤ) (h : sq = z * z) : square_number
```

Again, we can show that `4` is square using either `2` or `-2`:

```
lemma four_is_square :
  square_number 4 :=
square_number.mk 2 (by refl)

lemma four_is_square' :
  square_number 4 :=
square_number.mk (-2) (by refl)
```

The following lemma is one that we can only prove for `square_number`, even though we cannot prove the corresponding lemma for `square_roots`. We can show that the two definitions above are equal, as a consequence of proof irrelevance:

```
example : four_is_square = four_is_square' :=
by apply proof_irrel
```

Proof irrelevance forces the type `square_number 4` to contain only one element. Therefore, this element cannot contain the information as to whether it was constructed using `2` or `-2`. This is why large elimination on `square_number` would lead to an inconsistency and hence Lean will not allow us to define a large eliminator:

```
def large_elim' {sq : ℤ} {α : Sort v} :
  square_number sq → (∀(z : ℤ) (h : sq = z * z), α) → α
| (square_number.mk z h) f := f z h
-- induction tactic failed, recursor
-- 'LoVe.square_number.dcases_on' can only eliminate into Prop
```

But without being able to eliminate the `square_number` predicate at all, it would be quite useless. As a compromise, Lean allows us to define a *small eliminator*:

```
def small_elim {sq : ℤ} {α : Prop} :
  square_number sq → (∀(z : ℤ) (h : sq = z * z), α) → α
| (square_number.mk z h) f := f z h
```

It is called a small eliminator because it eliminates only into `Prop`, whereas a large eliminator can eliminate into an arbitrary large type universe `Sort v`. This eliminator cannot be used to prove that `four_is_square` and `four_is_square'` are different, but it can be used to prove that all square numbers are nonnegative, just like we could with the large eliminator:

```
example (sq : ℤ) (h : square_number sq) : sq ≥ 0 :=
begin
  apply small_elim h,
  intros z hz,
  rw hz,
  apply mul_self_nonneg
end
```

11.3 The Axiom of Choice

To understand the axiom of choice, it will help to first take a look at the following inductive predicate:

```
inductive nonempty (α : Sort u) : Prop
| intro (val : α) : nonempty
```

The predicate states that the given type α has at least one element. To prove `nonempty` α, we must provide an element of α to the `intro` rule:

```
lemma 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 from a proof of `nonempty` α.

The axiom of choice allows us to obtain some element of type α if we can show `nonempty` α:

```
axiom classical.choice {α : Sort u} : nonempty α → α
```

We have no way of knowing whether this is the element that was used to prove `nonempty` α. It will just give us an arbitrary element of α, thereby avoiding inconsistencies.

The constant `classical.choice` is noncomputable. If we ask Lean for its value using `#reduce` or `#eval`, it will refuse to compute it. This is one of the reasons why some logicians prefer to work without this axiom. Unlike proof irrelevance and the eliminators, this principle is not built into the Lean kernel; it is only an axiom in Lean's core library, giving users the freedom to work with or without it. Lean requires us to mark definitions as `noncomputable` if they use `choice` to define constants outside of `Prop`.

The following tools rely on `choice`:

- The function `classical.some` can help us find a witness of `∃a : α, p a`, if we do not care which one.

  ```
  classical.some : (∃a : α, p a) → α
  classical.some_spec (h : ∃a : α, p a) : p (some h)
  ```

- We can also derive the traditional axiom of choice:

  ```
  classical.axiom_of_choice : ∀α β {r : α → β → Prop},
    (∀x : α, ∃y : β, r x y) → (∃f, ∀x, r x (f x))
  ```

- The `choose` tactic provides an elegant way to use the set theoretic version of the axiom of choice in proofs:

  ```
  example (x : list ℕ)
      (h : ∀y : list ℕ, ∃z : list ℕ, [0] ++ z = x ++ y) :
    x.head = 0 :=
  begin
    choose z h using h,
    have hx : x = [0] ++ z [], by simp [h []],
    simp [hx]
  end
  ```

- Using `choice` and two other principles called propositional extensionality and functional extensionality, the law of excluded middle can be derived:

  ```
  classical.em : ∀p : Prop, p ∨ ¬ p
  ```

  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. The `by_cases` tactic offers a convenient interface for such proofs. To be able to use it with any proposition, add the line

  ```
  local attribute [instance, priority 0] classical.prop_decidable
  ```

  at the top of your file. Then you can use the `by_cases` tactic as follows:

  ```
  example (p : Prop) : p ∨ ¬ p :=
  begin
    by_cases h : p,
    { exact or.intro_left (¬ p) h, },
    { exact or.intro_right p h, },
  end
  ```

  In this example, the line `by_cases h : p` introduces a case distinction on whether `p` is true. It splits the goal into two subgoals, one with the additional assumption `h : p`, the other with the additional assumption `h : ¬ p`.

11.4 Subtypes

Subtyping is a mechanism to create new types from existing ones. Given a predicate on the elements of the original type, the new type contains only those elements of the original type that fulfill this property.

To illustrate this, we will define a type of full binary trees, building on the example from Section 4.3. In that section, we defined an inductive type `btree`, representing binary trees. In Section 5.4, we introduced a predicate `is_full` 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 `full_btree`, containing only full binary trees, as follows:

```
def full_btree (α : Type) :=
{ t : btree α // is_full t }
```

This is syntactic sugar for

```
def full_btree (α : Type) :=
@subtype (btree α) is_full
```

where `subtype` is defined as an inductive predicate:

```
inductive subtype {α : Type} (p : α → Prop) : Type
| mk : Πx : α, p x → subtype
```

Thus, every element of `full_btree` consists of a binary tree `t` and a proof that `t` is full:

```
def some_full_btree : full_btree ℕ :=
subtype.mk empty is_full.empty

def another_full_btree : full_btree ℕ :=
subtype.mk (node 6 empty empty)
  begin
    apply is_full.node,
    apply is_full.empty,
    apply is_full.empty,
    refl
  end
```

From a given element of `full_btree`, we can retrieve its two components via `subtype.val` and `subtype.property`. For example, `subtype.val another_full_btree` gives us the tree `node 6 empty empty` and `subtype.property another_full_btree` gives us a proof that this binary tree is full (i.e., `is_full (subtype.val another_full_btree)`). Using syntactic sugar, we can also write `another_full_btree.val` and `another_full_btree.property` instead.

We can transfer functions from the base type `btree` to the subtype `full_btree` if they preserve the property `is_full`. As we have shown in the lemma `is_full_mirror` in Section 5.4, `mirror` does preserve the property `is_full`. We can use that lemma to define a function `full_btree.mirror` operating on our new subtype as follows:

```
def full_btree.mirror {α : Type} (t : full_btree α) :
  full_btree α :=
subtype.mk (mirror t.val) (is_full_mirror t.val t.property)
```

The input of the function is an element `t` of our subtype `full_btree`. We decompose it into the binary tree `t.val` and the proof of being full `t.property`. We use the `mirror` function to mirror the tree component and the lemma `is_full_mirror` to transform the proof component into a proof that the mirrored tree is full. Finally, we recompose the elements using `subytpe.mk` into an element of our subtype.

For proofs about subtypes, the lemma `subtype.eq` is useful. It states that two elements of a subtype are equal if their `subtype.val` components are equal. Here is how it can be used to prove that the mirror image of a mirror image of a `full_btree` is the `full_btree` itself:

```
lemma full_btree.mirror_mirror {α : Type} (t : full_btree α) :
  t.mirror.mirror = t :=
begin
  apply subtype.eq,
  simp [full_btree.mirror],
  apply mirror_mirror
end
```

The lemma `subtype.eq` allows us to transform the goal into `mirror (mirror (t.val)) = t.val`, which matches the statement of the lemma `mirror_mirror` we proved earlier. The two `unfold` calls help understand the proof but are not necessary.

As a second example, consider the following definition of vectors from `mathlib`:

```
def vector (α : Type u) (n : ℕ) :=
{ l : list α // l.length = n }
```

Here, 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 is that some operations such as addition or scalar product of vectors rely on the two involved vectors having the same length.

Vectors can be built from lists using `subtype.mk` or the angle bracket notation:

```
def some_vector : vector ℤ 3 :=
subtype.mk ([1, 2, 3]) (by refl)
```

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 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 v.val)
  (begin rw list.length_map, exact v.property end)
```

We use `list.map` to negate every entry of the underlying list and `list.length_map` to show that this operation does not change the length of the list. Using `subtype.eq`, we can prove the following lemma about this new operator:

```
lemma vector.neg_neg (n : ℕ) (v : vector ℤ n) :
  vector.neg (vector.neg v) = v :=
begin
  apply subtype.eq,
  simp [vector.neg],
  rw [int.neg_comp_neg, list.map_id]
end
```

The application of `subtype.eq` reduces the goal to showing the corresponding property on the underlying lists. We can then use the lemmas `int.neg_comp_neg` and `list.map_id`, which state `int.neg ∘ int.neg = id` and `∀xs : list α, map id xs = xs`, respectively.

11.5 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. Just like subtypes, quotient types construct a new type from an existing type. Unlike a subtype, a quotient type contains all of the elements of the underlying type—but some elements that were different in the underlying type are considered equal in the quotient type.

The prerequisites to construct a quotient type are a type `base` and an equivalence relation `e : base → base → Prop` specifying which elements of `base` will be considered equal in the quotient. To construct the quotient type, we first need to declare `e` as an equivalence relation on `base`. A type with an equivalence relation is called a *setoid*. In Lean, `setoid` is a type class, of which we can declare an instance as follows:

```
instance base.setoid : setoid base :=
{ r := e,
  iseqv := sorry }
```

As a side effect, this instance declaration allows us to use the notation `a ≈ b` for `e a b`. More importantly, we can define the quotient type

```
def my_quotient : Type := quotient base.setoid
```

Every element `a` in `base` belongs to some element in `my_quotient`. We can find that element using

```
quotient.mk : base → my_quotient
```

As an abbreviation for `quotient.mk a`, we can also write `⟦a⟧`. The axiom `quotient.sound` guarantees that elements for which the relation `e` holds are indeed equal in the quotient type:

```
quotient.sound : ∀ {a b : base}, a ≈ b → ⟦a⟧ = ⟦b⟧
```

The lemma `quotient.exact` gives us the other direction:

```
quotient.exact : ∀ {a b : base}, ⟦a⟧ = ⟦b⟧ → a ≈ b
```

Finally, we can *lift* functions from `base → β` into `my_quotient → β` using `quotient.lift`, which has the following computation rule. Given some `f : base → β` such that `h : ∀a b, a ≈ b → f a = f b`, we have

```
quotient.lift f h ⟦a⟧ = f a
```

for all `a : base`. We can think of `quotient.lift` as an eliminator.

Construction of the Integers

As an example for a quotient type, we will now construct the integers. A construction that turns out to be very convenient is to construct a quotient over pairs of natural numbers (i.e., $\mathbb{N} \times \mathbb{N}$). The idea is that a pair `(m, n)` of natural numbers represents the integer `m - n`. In this way, we can represent all nonnegative integers `m` by `(m, 0)` and all negative integers `-n` by `(0, n)`. In addition, we get many more representations of the same integers, e.g., `(2, 1)`, `(3, 2)`, and `(4, 3)` all represent the integer `1`.

First, we need to register the equivalence relation that we want to use. We want two pairs `(k, l)` and `(m, n)` to be equal when `k - l` and `m - n` yield the same integer. However, the condition `k - l = m - n` does not work because the negation on $\mathbb{N}$ does not behave like the negation on integers, e.g., `0 - 1 = 0`. Instead, we use the condition `k + n = m + l`, which contains only addition.

We provide the definition of our equivalence relation, followed by a proof that it is indeed a equivalence relation (i.e., that it is reflexive, symmetric, and transitive):

```
instance int.rel : setoid (ℕ × ℕ) :=
{ r     := λa b, a.1 + b.2 = b.1 + a.2,
  iseqv :=
  begin
    apply and.intro,
    { intro a, refl },
    apply and.intro,
    { intros a b h, rw h },
    { intros a b c eq_ab eq_bc,
      apply eq_of_add_eq_add_right,
      calc (a.1 + c.2) + b.2 = (a.1 + b.2) + c.2 : by ac_refl
      ... = a.2 + (b.1 + c.2) : by rw [eq_ab]; ac_refl
      ... = (c.1 + a.2) + b.2 : by rw [eq_bc]; ac_refl }
  end }
```

We can now write $\approx$ for our equivalence relation:

```
lemma rel_iff (a b : ℕ × ℕ) :
  a ≈ b ↔ a.1 + b.2 = b.1 + a.2 :=
by refl
```

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.rel
```

We can define the integer zero as

```
def zero : int := ⟦(0, 0)⟧
```

In fact, any term of the form `⟦(n, n)⟧` represents the same integer:

```
example (n : ℕ) : zero = ⟦(n, n)⟧ :=
begin
  rw zero,
  apply quotient.sound,
  rw [rel_iff],
  simp
end
```

Next, we define addition on our newly constructed integers. To define functions on a quotient type, one can define the function on the original type first, and then *lift* the definition to the quotient type. To do that, 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 $\texttt{quotient.lift}_2$ (for binary functions) can be used to lift functions in this way.

Addition can be defined as simply adding the pairs of natural numbers componentwise. We then need provide a proof that this can be lifted to a function on the quotient by showing that $a_1 \approx b_1$ and $a_2 \approx b_2$ imply $[\![(a_1.1 + a_2.1, a_1.2 + a_2.2)]\!] = [\![(b_1.1 + b_2.1, b_1.2 + b_2.2)]\!]$:

```
def add : int → int → int :=
quotient.lift₂
  (λa b : ℕ × ℕ,
    ⟦(a.1 + b.1, a.2 + b.2)⟧)
  begin
    intros a₁ b₁ a₂ b₂ ha hb,
    apply quotient.sound,
    rw [rel_iff] at ha hb ⊢,
    calc (a₁.1 + b₁.1) + (a₂.2 + b₂.2)
        = (a₁.1 + a₂.2) + (b₁.1 + b₂.2) : by ac_refl
    ... = (a₂.1 + a₁.2) + (b₂.1 + b₁.2) : by rw [ha, hb]
    ... = (a₂.1 + b₂.1) + (a₁.2 + b₁.2) : by ac_refl
  end
```

This yields a function `add` with the intended behavior:

```
lemma add_mk (ap an bp bn : ℕ) :
  add ⟦(ap, an)⟧ ⟦(bp, bn)⟧ = ⟦(ap + bp, an + bn)⟧ :=
by refl
```

We can use this to prove other lemmas about `add`, such as

```
lemma add_zero (i : int) :
  add zero i = i :=
begin
  apply quotient.induction_on i,
  intro p,
  apply quotient.sound,
  simp
end
```

11.6 Summary of New Syntax

Tactics

`by_cases`	makes a case distinction on whether a given statement is true
`choose`	applies the axiom of choice to a hypothesis of the form ∀... ∃...

Constants

`classical.choice`	function that returns an arbitrary element of a nonempty type
`classical.some`	function that returns a witness given a proof of an existential
`Prop`	abbreviation for `Sort 0`
`quot`	function that creates a quotient type given an arbitrary relation
`quotient`	function that creates a quotient type of a given setoid instance
`setoid`	type class for a type with an equivalence relation on it
`Sort u`	type universe at level `u`
`Type u`	abbreviation for `Sort (u + 1)`

Notation

`{x : α // p x}`	subtype of all `x` in α fulfilling the predicate `p`
$\approx$	equivalence relation on a setoid

Chapter 12

Basic Mathematical Structures

In this chapter, we introduce definitions and proofs about basic mathematical structures.

12.1 Type Classes without Properties

A type class is a collection of abstract constants and their properties. A type can be declared an instance of a type class by providing concrete definitions for the constants and proving that the properties hold.

A simple example for a type class is the class `inhabited`, which requires only a constant `default_value` and no properties:

```
class inhabited (α : Type) :=
(default_value : α)
```

The type class `inhabited` is very similar to the inductive predicate `nonempty` that we discussed in Section 11.3. But since `inhabited` lives in `Type`, there is a large eliminator and we can extract the `default_value` that was used to construct an instance of the type class `inhabited` α.

Any type that has at least one element can be declared an instance of this class. For example, we can declare the natural numbers $\mathbb{N}$ as an instance by choosing an arbitrary natural number to be the default value:

```
instance : inhabited ℕ :=
{ default_value := 0 }
```

Using this type class, we can define a variant of the `head` operation on lists that we defined in Section 4.2. The issue with `head` is that there is no meaningful value that we can assign to `head []`. Given a type of type class `inhabited`, we can simply return the default value instead of using the `option` type.

```
def ihead {α : Type} [inhabited α] (xs : list α) : α :=
match xs with
| []     := inhabited.default_value α
| x :: _ := x
end
```

We require that the type α must be of type class `inhabited` by writing `[inhabited` α`]`. This allows us to access `inhabited.default_value` in the definition. The syntax `[inhabited` α`]` adds an implicit argument to the constant `ihead`. But unlike for other implicit arguments, Lean performs not only type inference but also a type class search through all declared instances to determine the value of this argument. Thus, when we write

```
#reduce ihead ([] : list ℕ)
```

Lean will search for an instance `inhabited` $\mathbb{N}$ and will find the instance we declared above. In that instance declaration, we defined `default_value` to be 0 and hence this is what the `#reduce` command returns.

We can prove abstract lemmas about the type class `inhabited` such as

```
lemma ihead_ihead {α : Type} [inhabited α] (xs : list α) :
  ihead [ihead xs] = ihead xs :=
```

For such lemmas, it is important to add the assumption `[inhabited` α`]` to be allowed to use the operator `ihead` 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 a few more type classes only requiring a constant but no properties—for example:

```
class has_zero (α : Type) := (zero : α)
class has_neg  (α : Type) := (neg : α → α)
class has_add  (α : Type) := (add : α → α → α)
class has_one  (α : Type) := (one : α)
class has_inv  (α : Type) := (inv : α → α)
class has_mul  (α : Type) := (mul : α → α → α)
```

These *syntactic* type classes introduce constants that are used in many 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 rich hierarchy of algebraic type classes and to allow overloading of common symbols such as $*$, 1, and $^{-1}$.

The syntactic type classes do not impose serious restrictions on the types that can be declared instances, except that a type of type class `has_zero` or `has_one` must be inhabited. In contrast, the *semantic* type classes, which we will discuss in the following sections, contain properties that restrict how the given constants behave.

12.2 Type Classes over a Single Binary Operator

The type classes `group` and `add_group` are examples of semantic type classes. In mathematics, a group is a set G with a a binary operator $\bullet : G \times G \to G$ fulfilling the following properties, called group axioms:

- Associativity: For all a, b, $c \in G$, we have $(a \bullet b) \bullet c = a \bullet (b \bullet c)$.
- Identity element: There exists an element $e \in G$ such that for all $a \in G$, we have $e \bullet a = a$.
- Inverse element: For each $a \in G$, there exists an inverse element denoted a^{-1} such that $a^{-1} \bullet a = e$.

Commonly, group operations are written multiplicatively (i.e., with operator $*$, identity element 1, and inverse element a^{-1}) or additively (i.e., 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 `add_group`. They are essentially the same but use different names for their constants and properties.

Any type that fulfills the group axioms can be instantiated as `group` or `add_group`. To illustrate this, we will define $\mathbb{Z}_2$, also known as $\mathbb{Z}/2\mathbb{Z}$, and instantiate it as an `add_group`. $\mathbb{Z}_2$ has two elements, which we will call `zero` and `one`:

```
inductive ℤ₂ : Type
| zero
| one
```

Addition is defined as follows:

```
def ℤ₂.add : ℤ₂ → ℤ₂ → ℤ₂
| ℤ₂.zero x       := x
| x       ℤ₂.zero := x
| ℤ₂.one  ℤ₂.one  := ℤ₂.zero
```

Invoking `#print add_group` tells us all the constants and properties we need to provide:

```
structure add_group : Type u → Type u
fields:
add_group.add : Π{α} [add_group α], α → α → α
add_group.add_assoc :
  ∀{α} [add_group α] (a b c : α), a + b + c = a + (b + c)
add_group.zero : Π(α) [add_group α], α
add_group.zero_add : ∀{α} [add_group α] (a : α), 0 + a = a
add_group.add_zero : ∀{α} [add_group α] (a : α), a + 0 = a
add_group.neg : Π{α} [c : add_group α], α → α
add_group.add_left_neg : ∀{α} [add_group α] (a : α), -a + a = 0
```

The constants `add_group.add`, `add_group.zero`, and `add_group.neg` respectively correspond to the binary operator •, the identity element *e*, and the inverse $^{-1}$ in the definition of a group above. The properties `add_group.add_assoc`, `add_group.zero_add`, and `add_group.add_left_neg` correspond to the three group axioms. Due to the way the `add_group` type class is constructed, we also need to provide the redundant property `add_group.add_zero`.

$\mathbb{Z}_2$ can be instantiated as a group as follows:

```
instance ℤ₂.add_group : add_group ℤ₂ :=
{ add          := ℤ₂.add,
  add_assoc    :=
    begin intros a b c, cases a; cases b; cases c; refl end,
  zero         := ℤ₂.zero,
  zero_add     := begin intro a, cases a; refl end,
  add_zero     := begin intro a, cases a; refl end,
  neg          := id,
  add_left_neg := begin intro a, cases a; refl end }
```

With this instance declaration, we can now use the notations `0`, `+`, and `-`:

```
#reduce ℤ₂.one + 0 - 0 - ℤ₂.one
```

Moreover, we get any lemma that has already been proved in general for the type class `add_group` for free:

```
example :
  ∀a : ℤ₂, a + - a = 0 :=
add_right_neg
```

The algebraic hierarchy contains some more type classes with one binary operator. Here is an overview of the most important ones:

Structure	Properties
`semigroup`	associativity
`monoid`	`semigroup` with unit (`1`)
`left_cancel_semigroup`	`semigroup` with `x * a = x * b → a = b`
`right_cancel_semigroup`	`semigroup` with `a * x = b * x → a = b`
`group`	`monoid` with inverse ($^{-1}$)

Most of these structures have commutative versions (where $a \bullet b = b \bullet a$ for all elements *a*, *b*), prefixed with `comm_`:

`comm_semigroup`, `comm_monoid`, `comm_group`

All of these type classes have additive counterparts with the prefix `add_`:

`add_semigroup`, `add_monoid`, `add_group`
`add_comm_semigroup`, `add_comm_monoid`, `add_comm_group`
`add_left_cancel_semigroup`, `add_right_cancel_semigroup`

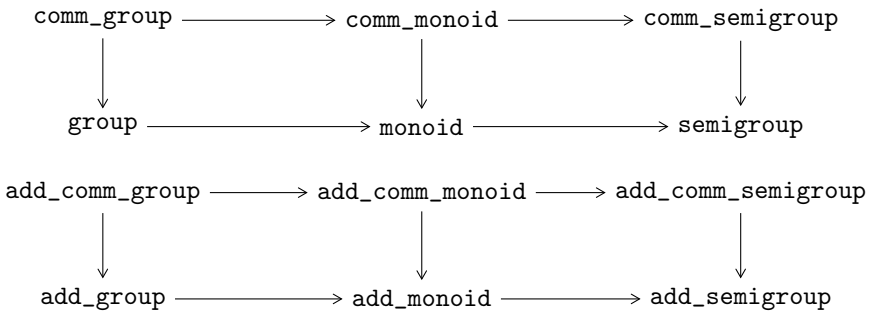
Although these additive type classes are isomorphic to their multiplicative counterparts, they are crucial when constructing algebraic structures with more than one binary operator such as rings and fields. To avoid duplicating all lemmas and definitions based on the multiplicative type classes, the copying process is automated by a tactic (`transport_multiplicative_to_additive`) and an attribute (`@[to_additive]`).

An example for an instance of `add_monoid` is the type `list` with the empty list `[]` as zero and the append operator ++ as addition:

```
instance {α : Type} : add_monoid (list α) :=
{ zero      := [],
  add       := (++),
  add_assoc := list.append_assoc,
  zero_add  := list.nil_append,
  add_zero  := list.append_nil }
```

We could go further an register this combination as a `add_left_cancel_semigroup` and an `add_right_cancel_semigroup`.

The graph below illustrates the relationships between some of these type classes. The arrows mean "inherits all properties from."



12.3 Type Classes over Two Binary Operators

The additive and multiplicative structures are amalgamated to form more complex type classes 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.
- $F\backslash\{0\}$ 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 invoking `#print field`, we can display all constants and properties required by the `field` type class. Again, the class includes some redundant properties such as `left_distrib` and `right_distrib`, due to the way in which it is constructed.

We will now show that $\mathbb{Z}_2$ is a field by instantiating the `field` type class with it. First, we define multiplication on $\mathbb{Z}_2$ as follows:

```
def ℤ₂.mul : ℤ₂ → ℤ₂ → ℤ₂
| ℤ₂.one  x       := x
| x       ℤ₂.one  := x
| ℤ₂.zero ℤ₂.zero := ℤ₂.zero
```

To declare $\mathbb{Z}_2$ a field, we can reuse the instance `ℤ₂.add_group` that we declared above using the syntax `..ℤ₂.add_group`. We can prove the remaining properties as follows:

```
instance : field ℤ₂ :=
{ one           := ℤ₂.one,
  mul           := ℤ₂.mul,
  inv           := id,
  add_comm      := begin intros a b, cases a; cases b; refl end,
  zero_ne_one   := begin intro h, cases h end,
  one_mul       := begin intros a, cases a; refl end,
  mul_one       := begin intros a, cases a; refl end,
  mul_inv_cancel :=
    begin intros a h, cases a, apply false.elim, apply h, refl, refl
    end,
  inv_mul_cancel :=
    begin intros a h, cases a, apply false.elim, apply h, refl, refl
    end,
  mul_assoc     :=
    begin intros a b c, cases a; cases b; cases c; refl end,
  mul_comm := begin intros a b, cases a; cases b; refl end,
  left_distrib  :=
    begin intros a b c, cases a; cases b; cases c; refl end,
  right_distrib :=
    begin intros a b c, cases a; cases b; cases c; refl end,
  ..ℤ₂.add_group }
```

With this declaration in place, we can now use the notations `1`, `*`, `/`, and more:

```
#reduce (1 : ℤ₂) * 0 / (0 - 1)  -- result: ℤ₂.zero
```

The type annotation `: ℤ₂` is necessary here to tell Lean that we want to calculate in $\mathbb{Z}_2$ and not in $\mathbb{N}$, the default. We can even use arbitrary numerals in $\mathbb{Z}_2$. For example, the numeral `3` is interpreted as `1 + 1 + 1`, which is the same as `1` in $\mathbb{Z}_2$:

```
#reduce (3 : ℤ₂)  -- result: ℤ₂.one
```

Moreover, we can use the tactic `ring` to normalize terms. For example:

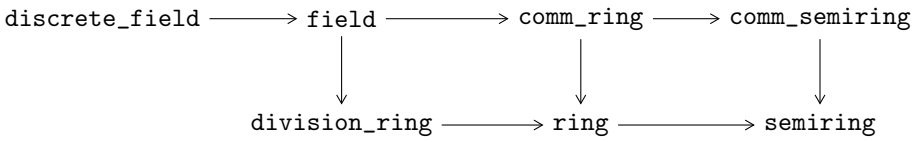
```
example (a b : ℤ₂) :
  (a + b) ^ 3 = a ^ 3 + 3 * a^2 * b + 3 * a * b ^ 2 + b ^ 3 :=
by ring -- normalizes terms of rings
```

This tactic is available for any type that is declared as a `field` or more generally as a `ring`. For commutative monoids and groups, there is a similar normalization tactic called `abel`.

Besides `field`, there are many more type classes for structures with two binary operators. These are the most important ones:

Structure	Properties	Examples
`semiring`	`monoid` and `add_comm_monoid` with distributivity	$\mathbb{R}$, $\mathbb{Q}$, $\mathbb{Z}$, $\mathbb{N}$
`comm_semiring`	`semiring` with commutativity of `*`	$\mathbb{R}$, $\mathbb{Q}$, $\mathbb{Z}$, $\mathbb{N}$
`ring`	`monoid` and `add_comm_group` with distributivity	$\mathbb{R}$, $\mathbb{Q}$, $\mathbb{Z}$
`comm_ring`	`ring` with commutativity of `*`	$\mathbb{R}$, $\mathbb{Q}$, $\mathbb{Z}$
`division_ring`	`ring` with multiplicative inverse	$\mathbb{R}$, $\mathbb{Q}$
`field`	`division_ring` with commutativity of `*`	$\mathbb{R}$, $\mathbb{Q}$
`discrete_field`	`field` with decidable equality and `∀n, n / 0 = 0`	$\mathbb{R}$, $\mathbb{Q}$

The property `∀n, n / 0 = 0` required by the `discrete_field` type class is simply a convention to make division a total function. Mathematicians would view division as a partial function.



The hierarchy between `ring` and `field` is more complex than shown in this overview and includes type classes for domains, integral domains, Euclidean rings, and more.

12.4 Coercions

When dealing with different kinds of numbers from $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$, and $\mathbb{R}$ at the same time, we may want to cast from one type into another. For example, given a natural number `n`, we may need to convert it to an integer:

```
lemma neg_mul_neg_nat (n : ℕ) (z : ℤ) :
  (- z) * (- n) = z * n :=
neg_mul_neg z n
```

Surprisingly, this statement does not lead to an error, although negation `- n` is not defined on `n : ℕ`, multiplication of `z : ℤ` with `n : ℕ` is not defined, and the lemma `neg_mul_neg` does not apply to natural numbers. By invoking `#print neg_mul_neg_nat`, we can see what happened:

```
theorem neg_mul_neg_nat : ∀(z : ℤ) (n : ℕ), -z * -↑n = z * ↑n :=
λ(z : ℤ) (n : ℕ), neg_mul_neg z ↑n
```

Lean has a mechanism to introduce coercions, represented by $\uparrow$ or `coe`, whenever necessary. This coercion operator can be set up to provide implicit conversions between arbitrary types. Many coercions are already in place, including the following:

- `coe` : $\mathbb{N} \to \alpha$ casts $\mathbb{N}$ into another semiring α;
- `coe` : $\mathbb{Z} \to \alpha$ casts $\mathbb{Z}$ into another ring α;
- `coe` : $\mathbb{Q} \to \alpha$ casts $\mathbb{Q}$ into another division ring α.

In some cases, Lean is unable to figure out where to place the coercions. We can then provide some type annotations to guide Lean's inference algorithm, as in the following example:

```
lemma neg_mul_neg_nat₂ (n : ℕ) (z : ℤ) :
  (- n : ℤ) * (- z) = n * z :=
neg_mul_neg n z
```

In proofs involving coercions, the tactic `norm_cast` can be convenient. It can help with proof goals such as

```
m n : ℕ,
h : ↑m = ↑n
⊢ m = n
```

in the proof

```
example (m n : ℕ) (h : (m : ℤ) = (n : ℤ)) :
  m = n :=
begin
  norm_cast at h,
  exact h
end
```

or

```
m n : ℕ
⊢ ↑m + ↑n = ↑(m + n)
```

in the proof

```
example (m n : ℕ) :
  (m : ℤ) + (n : ℤ) = ((m + n : ℕ) : ℤ) :=
by norm_cast
```

12.5 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 lemma, we should also reflect on alternatives such as multisets and finite sets.

Consider this definition based on the binary trees we introduced in Section 4.3:

```
def nodes_list : btree ℕ → list ℕ
| empty          := []
| (node a t₁ t₂) := a :: nodes_list t₁ ++ nodes_list t₂
```

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 in which order the elements occur in the tree. This is where multisets can help us. For multisets, we have `{3, 2, 1, 2}` = `{1, 2, 2, 3}`, whereas the 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 nodes_multiset : btree ℕ → multiset ℕ
| empty          := ∅
| (node a t₁ t₂) :=
  insert a (nodes_multiset t₁ ∪ nodes_multiset t₂)
```

Using this definition, we can prove lemmas such as `nodes_multiset t = nodes_multiset (mirror t)`, whereas `nodes_list t = nodes_list (mirror t)` does not hold.

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—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. We can redo the definition above using finsets as follows:

```
def nodes_finset : btree ℕ → finset ℕ
| empty          := ∅
| (node a t₁ t₂) := insert a (nodes_finset t₁ ∪ nodes_finset t₂)
```

For all three of these structures, Lean offers sum and product operators to add or multiply all of the values of a list, multiset, or finset:

```
#eval list.sum [1, 2, 3, 4]                       -- result: 10
#eval multiset.sum ({1, 2, 3, 4} : multiset ℕ)    -- result: 10
#eval finset.sum ({1, 2, 3, 4} : finset ℕ) id     -- result: 10

#eval list.prod [1, 2, 3, 4]                      -- result: 24
#eval multiset.prod ({1, 2, 3, 4} : multiset ℕ)   -- result: 24
#eval finset.prod ({1, 2, 3, 4} : finset ℕ) id    -- result: 24
```

The operators `finset.sum` and `finset.prod` take a second argument, a function that maps the values of the finset to the values that should be added or multiplied. If we put the

identity function `id` for that second argument, they behave like the corresponding operators for lists and multisets.

These operators require the type of the elements to be declared as an instance of `add_monoid` for addition or of `monoid` for multiplication. For `multiset` and `finset`, we additionally require an instance declaration for `add_comm_monoid` or `comm_monoid` because we cannot have the result depend on the order of adding or multiplying the elements.

`mathlib` provides a collections of lemmas describing how the big operators behave under reorderings, basic operators on `list`, `multiset`, and `finset`, homomorphisms, order relations, and more.

12.6 Order Type Classes

Many of the structures introduced above can be ordered. For example, the well-known 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 (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 fulfills the first three of these properties (reflexivity, transitivity, and antisymmetry), it is a *partial order*. An example of a partial order is the subset relation $\subseteq$ on sets, finite sets, or multisets. If a relation fulfills the first two of the above properties (reflexivity and transitivity), it is a *preorder*. An example of a preorder is comparing lists by their length.

In Lean, there are type classes for these different kinds of orders: `linear_order`, `partial_order`, and `preorder`. The `preorder` type class has the fields

```
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 `partial_order` type class has the additional field

```
le_antisymm : ∀a b : α, le a b → le b a → a = b
```

and `linear_order` has the additional field

```
le_total : ∀a b : α, le a b ∨ le b a
```

We can declare the preorder on lists that compares lists by their length as follows:

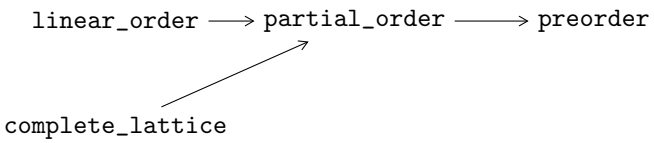
```
instance {α : Type} : preorder (list α) :=
{ le       := λx y, x.length ≤ y.length,
  le_refl  := λx, nat.le_refl x.length,
  le_trans := λx y z, nat.le_trans }
```

This type class instance gives us access to the infix syntax $\leq$ and to the corresponding relations $\geq$, <, and >:

```
example {α : Type} (c : α) :
  [c] > [] :=
dec_trivial
```

The proof uses the `dec_trivial` tactic, which attempts to decide a goal using an inferred decision procedure, relying on definitional equalities to evaluate the goal. This tactic tends to work well for expressions consisting only of closed terms.

Complete lattices, which we discussed in Chapter 10, are formalized as another type class `complete_lattice`, which is derived form `partial_order`. The arrows in the following graphical overview mean "inherits all properties from."



Type classes combining orders and algebraic structures are also available:

`ordered_cancel_comm_monoid`, `ordered_comm_group`
`ordered_semiring`, `linear_ordered_semiring`, `linear_ordered_comm_ring`
`linear_ordered_field`

All these mathematical structures relate $\leq$ and `<` with the constants `0`, `1`, `+`, and `*` by monotonicity rules (e.g., `a` $\leq$ `b` $\to$ `c` $\leq$ `d` $\to$ `a + c` $\leq$ `b + d`) and cancellation rules (e.g., `x + b` $\leq$ `x + c` $\to$ `b` $\leq$ `c`).

12.7 Summary of New Syntax

Tactics

`abel`	normalizes terms of commutative monoids and groups
`dec_trivial`	decides a goal (e.g., a true closed expression)
`norm_cast`	normalizes coercions
`ring`	normalizes ring terms

Chapter 13

Rational and Real Numbers

In the previous chapters, 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 present the construction of the rational numbers $\mathbb{Q}$ and the real numbers $\mathbb{R}$. The tools used for these constructions are the same as for $\mathbb{N}$ and $\mathbb{Z}$: inductive types and quotients.

A general recipe to construct new types with specific properties is as follows:

1. Create a new type that can represent all elements, but not necessarily in a unique manner.
2. Quotient this representation, equating elements that should be considered equal.
3. Define operators on the quotient type by lifting functions from the base type, and prove that they respect the quotient relation.

We have used this approach before to construct $\mathbb{Z}$. It can be used to construct $\mathbb{Q}$ and $\mathbb{R}$ as well.

13.1 Rational Numbers

A rational number is a number that can be expressed as a fraction n/d of integers n and $d \neq 0$:

```
structure fraction :=
(num           : ℤ)
(denom         : ℤ)
(denom_ne_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 from which we will derive a 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, meaning that $n_1 * d_2 = n_2 * d_1$. To construct the quotient of 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:

```
instance : setoid fraction :=
{ r     := λa b, a.num * b.denom = b.num * a.denom,
  iseqv :=
  begin
    apply and.intro,
```

```
    { intros a, simp },
    apply and.intro,
    { intros a b h, rw h },
    { intros a b c eq_ab eq_bc,
      apply eq_of_mul_eq_mul_right (fraction.denom_ne_zero b),
      calc (a.num * c.denom) * b.denom
          = (a.num * b.denom) * c.denom : by ac_refl
      ... = a.denom * (b.num * c.denom) : by rw [eq_ab]; ac_refl
      ... = (c.num * a.denom) * b.denom : by rw [eq_bc]; ac_refl }
  end}
```

Then we can define $\mathbb{Q}$ as the quotient over this setoid. To avoid a name clash, we call it $\mathtt{my}\mathbb{Q}$:

```
def myℚ : 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 the fractions to a common denominator and add the numerators. The easiest common denominator to use is simply the product of the two denominators:

```
instance : has_add fraction :=
{ add := λa b,
    { num           := a.num * b.denom + b.num * a.denom,
      denom         := a.denom * b.denom,
      denom_ne_zero :=
        mul_ne_zero a.denom_ne_zero b.denom_ne_zero } }
```

We register these operations directly as instances of the syntactic type classes such as `has_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 $\mathtt{my}\mathbb{Q}$, we first need to show that they respect the relation $\approx$:

```
@[simp] lemma add_num (a b : fraction) :
  (a + b).num = a.num * b.denom + b.num * a.denom :=
by refl

@[simp] lemma add_denom (a b : fraction) :
  (a + b).denom = a.denom * b.denom :=
by refl

lemma add_equiv_add {a b c d : fraction} (hac : a ≈ c)
    (hbd : b ≈ d) :
  a + b ≈ c + d :=
begin
  simp at hac hbd,
  have h₁ : a.num * b.denom * (c.denom * d.denom)
      = (c.num * a.denom) * b.denom * d.denom :=
    by rw ←hac; ac_refl,
  have h₂ : b.num * a.denom * (c.denom * d.denom)
      = (d.num * b.denom) * a.denom * c.denom :=
    by rw ←hbd; ac_refl,
  simp [add_mul, h₁, h₂],
  ac_refl
end
```

Then we can use `quotient.lift_on` and `quotient.lift_on₂` to define the operations on `myℚ`, and we can instantiate the relevant the syntactic type classes:

```
instance : has_add myℚ :=
{ add := λa b, quotient.lift_on₂ a b (λa b, ⟦a + b⟧)
    begin
      assume a b c d hac hbd,
      apply quotient.sound,
      exact fraction.add_equiv_add hac hbd
    end }
```

From here, we can proceed and prove all the properties needed to make `myℚ` an instance of `field` or, if we are more ambitious, `discrete_linear_ordered_field`.

Alternative Definitions of the Rational Numbers

We will discuss two more approaches to construct the rational numbers.

The first approach is the one used in `mathlib`. It is similar to the approach described above, but adds an additional property `cop : coprime num denom` to the structure, which states that that numerator and denominator do not have a common divisor (except `1` and `-1`):

```
structure rat :=
(num : ℤ)
(denom : ℕ)
(pos : 0 < denom)
(cop : coprime num denom)
```

On the positive side, no quotient construction is required, computation may be more efficient, and more theorems are definitional equalities. A disadvantage is that function definitions become more complicated.

A second alternative approach is to define all elements syntactically, including the desired operations:

```
inductive pre_rat : Type
| zero : pre_rat
| one : pre_rat
| add : pre_rat → pre_rat → pre_rat
| sub : pre_rat → pre_rat → pre_rat
| mul : pre_rat → pre_rat → pre_rat
| div : pre_rat → pre_rat → pre_rat
```

Then we define an equivalence relation and take its quotient to enforce congruence rules and the field axioms.

```
inductive equiv : pre_rat → pre_rat → Prop
| add_congr {a b c d : pre_rat} :
    equiv a b → equiv c d → equiv (add a c) (add b d)
| add_assoc {a b c : pre_rat} :
    equiv (add a (add b c)) (add (add a b) c)
| zero_add {a : pre_rat} : equiv (add zero a) a
| add_comm {a b : pre_rat} : equiv (add a b) (add b a)
| etc : equiv sorry sorry

def rat : Type :=
quot equiv
```

The advantages are that this construction does not require `ℤ` and that it is easy to declare as an instance of `field`. Most importantly, though, this approach can be used for other

algebraic constructions, such as free monoids and free groups. A disadvantage is that the definition of orders and lemmas about them become more complicated.

13.2 Real Numbers

There are sequences of rational numbers that 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

$$\begin{aligned} a_0 &= 1 \\ a_1 &= 1.4 \\ a_2 &= 1.41 \\ a_3 &= 1.414 \\ a_4 &= 1.4142 \\ a_5 &= 1.41421 \\ a_6 &= 1.414213 \\ a_7 &= 1.4142135 \\ a_8 &= 1.41421356 \\ &\vdots \end{aligned}$$

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 $\sqrt{2}$, which is not a rational number.

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 do not deviate more 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 is_cau_seq (f : ℕ → ℚ) : Prop :=
∀ε > 0, ∃N, ∀m ≥ N, abs (f N - f m) < ε
```

We define a type of Cauchy sequences as a subtype:

```
def cau_seq : Type :=
{f : ℕ → ℚ // is_cau_seq f}
```

The idea of this construction is to let the real numbers be represented 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. Formally, two sequences represent the same real number when their difference converges to zero:

```
instance equiv : setoid cau_seq :=
{ r := λf g, ∀ε > 0, ∃N, ∀m ≥ N, abs (f.val m - g.val m) < ε,
  iseqv := sorry }
```

We omitted the proof that this is an equivalence relation using `sorry` for brevity. Using this `setoid` instance, we can now define the real numbers:

```
def my_real : Type :=
quotient cau_seq.equiv
```

Like for the rational numbers, we need to define zero, one, addition, multiplication, and other operators. We define them on `cau_seq` first and lift them to `my_real` afterwards. For the constants `0` and `1`, we can define them simply as a constant sequence. Any constant sequence is a Cauchy sequence:

```
def const (x : ℚ) : cau_seq :=
subtype.mk (λ_, x) (begin intros ε hε, use 0, simpa using hε end)
```

We can declare `my_real` instances of the syntactic type classes `has_zero` and `has_one` as follows:

```
instance : has_zero my_real :=
{ zero := ⟦cau_seq.const 0⟧ }

instance : has_one my_real :=
{ one := ⟦cau_seq.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 : has_add cau_seq :=
{ add := λf g, subtype.mk (λn, f.val n + g.val n) sorry}
```

This definition requires a proof that the result is a Cauchy sequence, given that `f` and `g` are Cauchy sequences. We omit that proof here. Next, we need to show that this addition is compatible with our equivalence relation:

```
lemma add_equiv_add {f₁ f₂ g₁ g₂ : cau_seq}  (hf : f₁ ≈ f₂)
    (hg : g₁ ≈ g₂) :
  f₁ + g₁ ≈ f₂ + g₂ :=
begin
  intros ε₀ hε₀,
  cases hf (ε₀ / 2) (half_pos hε₀) with Nf hNf,
  cases hg (ε₀ / 2) (half_pos hε₀) with Ng hNg,
  use max Nf Ng,
  intros m hm,
  calc abs ((f₁ + g₁).val m - (f₂ + g₂).val m)
      = abs ((f₁.val m + g₁.val m) - (f₂.val m + g₂.val m)) :
    by refl
  ... = abs ((f₁.val m - f₂.val m) + (g₁.val m - g₂.val m)) :
    by simp
  ... ≤ abs (f₁.val m - f₂.val m) + abs (g₁.val m - g₂.val 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
```

We will discuss the proof in detail. It has a similar flavor to the two proofs we omitted above. To show that $f_1 + g_1 \approx f_2 + g_2$, we are given an $\varepsilon_0 > 0$ and must prove that there exists an $N : \mathbb{N}$ such that

$$\forall m : \mathbb{N},\ m \geq N \to \text{abs}\ ((f_1 + g_1).\text{val}\ m - (f_2 + g_2).\text{val}\ m) < \varepsilon_0$$

To obtain such an `N`, we use the facts $f_1 \approx f_2$ and $g_1 \approx g_2$. The fact $f_1 \approx f_2$ gives us for any $\varepsilon > 0$ a number `Nf` such that $\text{abs}\ (f_1.\text{val}\ m - f_2.\text{val}\ m) < \varepsilon$ for all $m \geq Nf$. The fact $g_1 \approx g_2$ 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 := ε_0 / 2. Then we choose `N` to be the maximum of `Nf` and `Ng`, so that we get the inequalities for any $\mathtt{m} \geq \mathtt{N}$. The `calc` block at the end of the proof establishes that `abs ((f₁ + g₁).val m - (f₂ + g₂).val m) < ε₀` for all $\mathtt{m} \geq \mathtt{N}$.

Using the lemma, we can then define addition on `my_real`:

```
instance : has_add my_real :=
{ add := λa b, quotient.lift_on₂ a b (λa b, ⟦a + b⟧)
    begin
      assume a b c d hac hbd,
      apply quotient.sound,
      exact cau_seq.add_equiv_add hac hbd,
    end }
```

Multiplication can be defined similarly.

Alternative Definitions of the Real Numbers

In `mathlib`, the construction of the real numbers is essentially as described above. Only some definitions are stated in a more general fashion to allow construction of other algebraic structures, such as the *p*-adic numbers.

Alternatively, the real numbers can be defined using Dedekind cuts. Essentially, a number $\mathtt{r} : \mathbb{R}$ is then represented as $\{\mathtt{x} : \mathbb{Q} \mid \mathtt{x} < \mathtt{r}\}$.

Another alternative definition of the reals is to define them using binary sequences $\mathbb{N} \to \mathtt{bool}$. The elements of the sequence represent the digits of the real number. This works particularly well if we only need the real numbers in the interval $[0, 1]$. This construction does not require the rational numbers.

Bibliography

[1] J. Avigad, L. de Moura, and S. Kong. *Theorem Proving in Lean: Release 3.4.0*. 2019. https://leanprover.github.io/theorem_proving_in_lean/.

[2] J. Avigad, L. de Moura, and J. Roesch. *Programming in Lean*. 2016. https://leanprover.github.io/programming_in_lean/.

[3] J. Avigad, G. Ebner, and S. Ullrich. *The Lean Reference Manual: Release 3.3.0*. 2018. https://leanprover.github.io/reference/.

[4] H. P. Barendregt, W. Dekkers, and R. Statman. *Lambda Calculus with Types*. Perspectives in logic. Cambridge University Press, 2013.

[5] G. Ebner, S. Ullrich, J. Roesch, J. Avigad, and L. de Moura. A metaprogramming framework for formal verification. *PACMPL*, 1(ICFP):34:1–34:29, 2017.

[6] G. Gonthier. Formal proof—The four-color theorem. *Notices AMS*, 55(11):1382–1393, 2008.

[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, *Interactive Theorem Proving (ITP 2013)*, volume 7998 of *Lecture Notes in Computer Science*, pages 163–179. Springer, 2013.

[8] 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, *Operating Systems Design and Implementation (OSDI 2016)*, pages 653–669. USENIX Association, 2016.

[9] 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. Available at http://arxiv.org/abs/1501.02155.

[10] J. Harrison. Formal verification at Intel. In *Logic in Computer Science (LICS 2003)*, pages 45–54. IEEE Computer Society, 2003.

[11] J. R. Hindley. The principal type-scheme of an object in combinatory logic. *Trans. Amer. Math. Soc.*, (146):29–60, 1969.

[12] 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.

[13] X. Leroy. A formally verified compiler back-end. *J. Automated Reasoning*, 43(4):363–446, 2009.

[14] A. Lochbihler. Verifying a compiler for Java threads. In A. D. Gordon, editor, *European Symposium on Programming (ESOP 2010)*, volume 6012 of *Lecture Notes in Computer Science*, pages 427–447. Springer, 2010.

[15] R. Milner. A theory of type polymorphism in programming. *J. Comput. Syst. Sci.*, 17(3):348–375, 1978.

[16] R. Nederpelt and H. Geuvers. *Type Theory and Formal Proof: An Introduction*. Cambridge University Press, 2014.

[17] T. Nipkow and G. Klein. *Concrete Semantics—With Isabelle/HOL*. Springer, 2014.

[18] B. O'Sullivan, D. Stewart, and J. Goerzen. *Real World Haskell*. O'Reilly, 2008. `http://book.realworldhaskell.org/read/`.

[19] A. J. Perlis. Epigrams on programming. *SIGPLAN Notices*, 17(9):7–13, 1982.

[20] B. C. Pierce. *Types and programming languages*. MIT Press, 2002.

[21] 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.

[22] F. van Raamsdonk. *Logical Verification: Course Notes*. 2011. `https://www.cs.vu.nl/~jbe248/lv2017/notes.pdf`.