Anne Baanen
Alexander Bentkamp
Jasmin Blanchette
Johannes Hölzl

The Hitchhiker's Guide to Logical Verification

2020 Edition
(April 12, 2020)

lean-forward.github.io/
logical-verification/2020

Contents

Contents iii

Preface vii

I Basics 1

1 Definitions and Statements 3
1.1 Types and Terms . . . 3
1.2 Type Definitions . . . 8
1.3 Function Definitions . . . 13
1.4 Lemma Statements . . . 16
1.5 Summary of New Constructs . . . 18

2 Backward Proofs 19
2.1 Tactic Mode . . . 19
2.2 Basic Tactics . . . 21
2.3 Reasoning about Connectives and Quantifiers . . . 23
2.4 Reasoning about Equality . . . 27
2.5 Rewriting Tactics . . . 28
2.6 Proofs by Mathematical Induction . . . 29
2.7 Induction Tactic . . . 30
2.8 Cleanup Tactics . . . 31
2.9 Summary of New Constructs . . . 31

3 Forward Proofs 33
3.1 Structured Proofs . . . 33
3.2 Structured Constructs . . . 35
3.3 Forward Reasoning about Connectives and Quantifiers . . . 37
3.4 Calculational Proofs . . . 39
3.5 Forward Tactics . . . 40
3.6 Dependent Types . . . 41
3.7 The Curry–Howard Correspondence . . . 43
3.8 Induction by Pattern Matching . . . 45
3.9 Summary of New Constructs . . . 47

II Functional–Logic Programming **49**

4 Functional Programming **51**
4.1 Inductive Types . . . 51
4.2 Structural Induction . . . 52
4.3 Structural Recursion . . . 54
4.4 Pattern Matching Expressions . . . 55
4.5 Structures . . . 56
4.6 Type Classes . . . 58
4.7 Lists . . . 62
4.8 Binary Trees . . . 67
4.9 Case Distinction and Induction Tactics . . . 68
4.10 Dependent Inductive Types . . . 70
4.11 Summary of New Constructs . . . 72

5 Inductive Predicates **73**
5.1 Introductory Examples . . . 73
5.2 Logical Symbols . . . 78
5.3 Rule Induction . . . 79
5.4 Rule Induction Pitfalls . . . 81
5.5 Miscellaneous Tactics . . . 83
5.6 Elimination . . . 83
5.7 Further Examples . . . 86
5.8 Summary of New Constructs . . . 90

6 Monads **91**
6.1 Introductory Example . . . 91
6.2 Two Operations and Three Laws . . . 93
6.3 A Type Class . . . 95
6.4 Identity . . . 96
6.5 Exceptions . . . 97
6.6 Mutable State . . . 99
6.7 Nondeterminism . . . 102
6.8 Tautology Tactic . . . 103
6.9 A Generic Algorithm: Iteration over a List . . . 103
6.10 Summary of New Constructs . . . 104

7 Metaprogramming **105**
7.1 Tactics and Tactic Combinators . . . 106
7.2 The Metaprogramming Monad . . . 109
7.3 Names, Expressions, Declarations, and Environments . . . 112
7.4 First Example: A Conjunction-Destructing Tactic . . . 116
7.5 Second Example: A Provability Advisor . . . 118
7.6 A Look at Two Predefined Tactics . . . 119
7.7 Miscellaneous Tactics . . . 120
7.8 Summary of New Constructs . . . 121

III Program Semantics **123**

8 Operational Semantics **125**
8.1 Formal Semantics . . . 125
8.2 A Minimalistic Imperative Language . . . 126
8.3 Big-Step Semantics . . . 127
8.4 Properties of the Big-Step Semantics . . . 129
8.5 Small-Step Semantics . . . 131
8.6 Properties of the Small-Step Semantics . . . 133
8.7 Parallelism . . . 134

9 Hoare Logic **137**
9.1 Hoare Triples . . . 137
9.2 Hoare Rules . . . 138
9.3 A Semantic Approach to Hoare Logic . . . 140
9.4 First Program: Exchanging Two Variables . . . 142
9.5 Second Program: Adding Two Numbers . . . 143
9.6 Finish Tactic . . . 145
9.7 A Verification Condition Generator . . . 145
9.8 Second Program Revisited: Adding Two Numbers . . . 147
9.9 Hoare Triples for Total Correctness . . . 147
9.10 Summary of New Constructs . . . 148

10 Denotational Semantics **149**
10.1 Compositionality . . . 149
10.2 A Relational Denotational Semantics . . . 150
10.3 Fixpoints . . . 151
10.4 Monotone Functions . . . 152
10.5 Complete Lattices . . . 152
10.6 Least Fixpoint . . . 154
10.7 A Relational Denotational Semantics, Continued . . . 154
10.8 Application to Program Equivalence . . . 154

IV Mathematics **157**

11 Logical Foundations of Mathematics **159**
11.1 Universes . . . 159
11.2 The Peculiarities of Prop . . . 160
11.3 The Axiom of Choice . . . 162
11.4 Subtypes . . . 163
11.5 Quotient Types . . . 166
11.6 Summary of New Constructs . . . 171

12 Basic Mathematical Structures **173**
12.1 Type Classes over a Single Binary Operator . . . 173
12.2 Type Classes over Two Binary Operators . . . 176
12.3 Coercions . . . 178
12.4 Normalization Tactics . . . 179
12.5 Lists, Multisets, and Finite Sets . . . 179

12.6 Order Type Classes 181
12.7 Summary of New Constructs 182

13 Rational and Real Numbers **183**
13.1 Rational Numbers 183
13.2 Real Numbers 186
13.3 Summary of New Constructs 190

Bibliography **191**

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. In this context, "formal" means "logical" or "logic-based." Logicians—the mathematicians of logics—carried out formal proofs on papers decades before the advent of computers, but nowadays formal proofs are almost always carried out using a proof assistant.

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

The main strength of proof assistants is that they help develop highly trustworthy, unambiguous proofs of mathematical statements, using a precise logic. They can be used to prove arbitrarily advanced results, not only toy examples or logic puzzles. Formal proofs also help students understand what constitutes a valid definition or a valid proof. To quote Scott Aaronson:[1]

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

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

Success Stories There have been a number of 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. [10]. The earliest work in this area was carried out by Nicolaas de Bruijn and his colleagues starting in the 1960s in a system called AUTOMATH.[2]

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

Most users of proof assistants today are computer scientists. A few companies, including AMD [29] and Intel [11], have been using proof assistants to verify their designs. In academia, some of the milestones are the verifications of the operating system kernels seL4 [14] and CertiKOS [9] and the development of the verified compilers CompCert [17], JinjaThreads [20], and CakeML [16].

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

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

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

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

We will use community version 3.5.1, which is expected to be one of the last releases before Lean 4. We will use Lean's basic libraries, revision `2d6556dd` of `mathlib`, and a few extensions collected in a small library called `LoVelib`.

Although it is still a research project, with some rough edges, there are several reasons why Lean is a suitable vehicle to teach interactive theorem proving:

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

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

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

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

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

This Guide This guide is a companion to the MSc-level course Logical Verification (LoVe) taught at the Vrije Universiteit Amsterdam. Our primary aim is to teach interactive theorem proving. Lean is the vehicle, not an end of itself. As such, this guide is not designed to be a comprehensive Lean tutorial—for this, we recommend *Theorem Proving in Lean* [1]. The guide is also no substitute for attending class or doing the exercises and homework. Theorem proving is not for spectators; it can only be learned by doing.

Specifically, our goal is that you

- learn fundamental theory and techniques in interactive theorem proving;
- familiarize yourselves with some areas in which proof assistants are successfully applied, such as functional programming, the semantics of imperative programming languages, and mathematics;
- develop some practical skills which you can apply on a larger project (as a hobby, for an MSc or PhD, or in industry);
- reach a point where you feel ready to move to another proof assistant and apply what you have learned;
- get to understand the domain well enough to start reading relevant scientific papers published at international conferences such as Certified Programs and Proofs (CPP) or Interactive Theorem Proving (ITP) or in journals such as the *Journal of Automated Reasoning* (JAR).

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

We assume that you are familiar with typed functional programming, as embodied in the languages Haskell, OCaml, and Standard ML. If you do not see that the term `g (f a b)` applies the (curried) function `f` to two arguments `a` and `b` and passes the result to the function `g`, or that `λn, n + 1` is the function that maps `n` to `n + 1`, we strongly recommend that you start by studying a tutorial, such as the first chapters of the online tutorial *Learn You a Haskell for Great Good!* [19]. You can stop reading once you have reached the end of the section titled "Lambdas."

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

The Lean files accompanying this guide can be found in a public repository.[5] The files' naming scheme follows this guide'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* [24], who have taught us Lean and programming language semantics. Many of their ideas appear in this guide.

We thank Robert Lewis and Assia Mahboubi for their useful comments on this guide's organization and focus. We thank Kiran Gopinathan and Ilya Sergey for

[5] `https://github.com/blanchette/logical_verification_2020`

sharing the anecdote reported in footnote 2 of Chapter 1 and letting us share it further. Finally, we thank Kevin Buzzard and Wijnand van Woerkom for reporting typos and some more serious errors they found in an earlier edition of this guide. Should *you* find some mistakes in this text, please let us know.

Special Symbols Visual Studio Code lets us enter Unicode symbols by entering backslash \ followed by an ASCII identifier. For example, $\to$, $\mathbb{Z}$, or $\mathbb{Q}$ can be entered by typing `\->`, `\Z`, or `\Q` 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 guide and, for each, one of its ASCII representations. By hovering over a symbol in Visual Studio Code while holding the control or command key pressed, you will see the different ways in which it can be entered.

$\neg$	`\not`	$\wedge$	`\and`	$\vee$	`\or`
$\to$	`\->`	$\leftrightarrow$	`\<->`	$\forall$	`\fo`
$\exists$	`\ex`	$\leq$	`\<=`	$\geq$	`\>=`
$\neq$	`\neq`	$\approx$	`\~~`	$\times$	`\x`
$\circ$	`\circ`	$\emptyset$	`\empty`	$\cup$	`\union`
$\cap$	`\intersect`	$\in$	`\in`	$\downharpoonleft$	`\downleftharpoon`
$\bigcirc$	`\bigcirc`	$\leftarrow$	`\<-`	$\mapsto$	`\mapsto`
$\Rightarrow$	`\=>`	$\Longrightarrow$	`\==>`	$[\![$	`\[[`
$]\!]$	`\]]`	$\mathcal{A}$	`\McA`	$\mathbb{N}$	`\N`
$\mathbb{Z}$	`\Z`	$\mathbb{Q}$	`\Q`	$\mathbb{R}$	`\R`
α	`\a`	β	`\b`	γ	`\g`
δ	`\de`	ε	`\e`	λ	`\la`
σ	`\s`	$_0$	`\0`	$_1$	`\1`
$_2$	`\2`	$_3$	`\3`	$_4$	`\4`
$_5$	`\5`	$_6$	`\6`	$_7$	`\7`
$_8$	`\8`	$_9$	`\9`		

Part I

Basics

Chapter 1

Definitions and Statements

We start our journey by studying the basics of Lean, without carrying out actual proofs yet. We review how to define custom 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:

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

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

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

closely corresponds to the mathematical definition

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

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. It can also be viewed as a generalization of first-order logic (also called predicate logic).

1.1.1 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. Lean's types may be *interpreted* as 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 $(\mathbb{Z} \to \mathbb{Z}) \to \mathbb{Q}$ is the type of unary functions that take a function of type $\mathbb{Z} \to \mathbb{Z}$ as argument and that return a value of type $\mathbb{Q}$.

1.1.2 Terms

The *terms*, or expressions, of simple type theory consist of constants `c`, variables `x`, applications `t u`, and λ-expressions λ`x, t`, 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 global 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`. Functions (e.g., `abs`) and predicates (e.g., `prime`) are considered constants.

A *variable* `x : `σ is either *bound* or *free*. A bound variable 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`. A free variable is declared in the local context—a concept that will be explained below.

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., `abs 0`. No parentheses are needed 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)`, `1` to `square (abs 1)`, and so on. A more mathematician-friendly syntax would have been `x` $\mapsto$ `square (abs x)`, but this is not supported by Lean.

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 an ingenious 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 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 :=` (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 to a single argument, is said to be *partially applied*.

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

```
σ → τ → υ   for   σ → (τ → υ)
    t u v   for   (t u) v
λx : σ, λy : τ, t   for   λx : σ, (λy : τ, t)
```

and also

```
λ(x : σ) (y : τ), t   for   λx : σ, λy : τ, t
      λx y : σ, t   for   λ(x : σ) (y : σ), t
```

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 `λx, (+) 1 x` and `λx, 1 + x`.

One way to work with Lean is to declare the types and constants we need using the `constant(s)` 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` command to type-check some terms and print their types. The `#` prefix identifies diagnosis commands: commands that are useful for debugging but that we would normally not keep in our Lean files.

The `constant(s)` command can even be used to declare types. In Lean, types are terms like any other. The next example introduces a type of "Trooleans" and three constants of that type:

```
constant trool : Type
constants trool.true trool.false trool.maybe : trool
```

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

1.1.3 Type Checking and Type Inference

When Lean parses a term, it checks whether the term is well typed. In the process, it will try to infer the types of bound variables if those are omitted—e.g., the type of `x` in `λx, 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 : ℕ` and `0 : ℝ`) can lead to undecidability [27]. Lean takes a pragmatic approach and assumes that numerals `0`, `1`, `2`, ..., are of type `ℕ` if several types are possible.

Lean's type system can be expressed as a formal system. A *formal system* consists of *judgments* and of (*derivation*) *rules* for producing judgments. A typing judgment has the form $\mathtt{C} \vdash \mathtt{t} : \sigma$, meaning that term $\mathtt{t}$ has type σ in local context $\mathtt{C}$. The local context gives the types of the variables in $\mathtt{t}$ that are not bound by any λ. For simple type theory, there are four typing rules, corresponding to the four kinds of terms:

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

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

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

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

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

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

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

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

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

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 [13] and Robin Milner [22], which also forms the basis of Haskell, OCaml, and Standard ML. The algorithm generates type constraints involving type variables $?\alpha$, $?\beta$, $?\gamma$, ..., and attempts to solve them using a type unification procedure. For example, when

inferring the type $?\alpha$ of $\lambda x,\ \texttt{abs}\ x$, Lean would perform the following schematic type derivation:

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

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

1. For the application of Lam, the type of $\lambda x,\ \texttt{abs}\ x$ must be of the form $?\beta \to ?\gamma$, for some types $?\beta$ and $?\gamma$. Thus, Lean would generate the constraint $?\alpha = ?\beta \to ?\gamma$.
2. For the application of Cst, the type of $\texttt{abs}$ must correspond to the declaration as $\mathbb{Z} \to \mathbb{N}$. Thus, Lean would generate the constraint $?\beta \to ?\gamma = \mathbb{Z} \to \mathbb{N}$.

Solving the two constraints yields $?\alpha := \mathbb{Z} \to \mathbb{N}$, which is indeed the type that Lean infers for $\lambda x,\ \texttt{abs}\ x$.

1.1.4 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 "provide a term of type σ" are good practice towards mastery of theorem proving.

To create a term of a given type, start with the placeholder _ and recursively apply a combination of the following two steps:

1. If the type is of the form $\sigma \to \tau$, a possible inhabitant is an anonymous function, of the form $\lambda x,\ \_$, 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 variables declared in the local context.
2. Given a type σ (which may be a function type), you can use any constant c or variable $x : \tau_1 \to \cdots \to \tau_n \to \sigma$ to build a term of that type. For each argument, you need to put a placeholder, yielding $c\ \_ \ldots \_$ or $x\ \_ \ldots \_$.

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

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

Recall that we are trying to build a term of type γ. The only variable we can use to achieve this is `f`: It takes two arguments and returns a value of type γ. So we replace the placeholder with the term `f _ _`, where the two new placeholders stand for the two missing arguments. Putting everything together, we now have the term `λf, λg, λa, 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`, of type α, with no arguments. For the second placeholder, we apply step 2 with the variable `g`, which is the only source of βs. Since `g` takes an argument, we must supply a placeholder. This means our current term is `λf, λg, λa, f a (g _)`.

We are almost done. The remaining placeholder has type $\beta \to \alpha$, corresponding to `g`'s argument type. Applying step 1, we replace the placeholder with `λb, _`, where the new placeholder has type α. Here, we can supply `a`. Our final term is `λf, λg, λa, f a (g (λb, a))`—i.e., `λf 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. For some other types, we might encounter dead ends and need to backtrack. We might also fail altogether, with nowhere to backtrack to. After all, with an empty local context, it is impossible to supply a witness for α.

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

1.2 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 are built by applying special constants called *constructors*. Inductive types are a concise way of representing acyclic data in a program. You may know them under some other, largely equivalent names, including algebraic data types, inductive data types, freely generated data types, recursive data types, and data types.

1.2.1 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)
⋮
```

denote the different values of type `nat`—zero, its successor, its successor's successor, and so on. This notation is called *unary*, or *Peano*, after the logician Giuseppe Peano. For an alternative explanation of Peano numbers in Lean (and some groovy video game graphics), we refer to Kevin Buzzard's article "Can computers prove theorems?"[1]

The general format of type declarations is

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

For the natural numbers, it is possible to convince Lean to use the familiar names ℕ, `0`, `1`, `2`, ..., and indeed the predefined `nat` type offers such syntactic sugar. But using the lengthier notations sheds more light on Lean's type definitions. Besides, syntactic sugar is known to cause cancer of the semicolon [26].

In the Lean file accompanying this chapter, the definition of `nat` is located in 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` will refer to Lean's predefined type of natural numbers. Similarly, the entire file is located 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` namespace displays the following information:

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

The focus on natural numbers is one of the many features of this guide that reveal a bias towards computer science. Number theorists would be more interested in the integers ℤ and the rational numbers ℚ; analysts would want to work with the real numbers ℝ and the complex numbers ℂ. But the natural numbers are ubiquitous in computer science and enjoy a very simple definition as an inductive type. They can also be used to build other types, as we will see in Chapter 13.

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

1.2.2 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 `aexp.num i` is an `aexp` value.
2. For every character string `x`, we have that `aexp.var x` is an `aexp` value.
3. If e_1 and e_2 are `aexp` values, then so are `aexp.add` e_1 e_2, `aexp.sub` e_1 e_2, `aexp.mul` e_1 e_2, and `aexp.div` e_1 e_2.

The above definition is exhaustive. The only possible values for `aexp` are those built using formation rules 1 to 3. Moreover, `aexp` values 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.

1.2.3 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.

1.2.4 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, `aexp.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 to allocate exactly the right amount of memory.

1.2.5 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 as a *parameter* 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 argument 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 Πα `: Type,` $\alpha \to$ `list` $\alpha \to$ `list` α.

Even if we try to restrict ourselves to a fragment of Lean's language, Lean often exposes us to more advanced constructs in the output, such as Π above, `Sort u`, or `Type 1`. Our advice is to adopt a *sporty attitude*: Do not worry if you do not always understand everything the first time. Use your common sense and your imagination. And, above all, do not hesitate to ask.

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

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

The `::` operator, like all other binary operators, binds less tightly than function application. Thus, `f x :: 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`.

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

1.3 Function Definitions

If all we want is to declare a function, we can use the `constant` command described above (Section 1.1.2). 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 definitions by pattern matching is

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

The parentheses `( )` around the parameters can also be curly braces `{ }` if we want to make them implicit arguments. 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 (params₁ : type₁) ... (paramsₘ : typeₘ) : type :=
result
```

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

The basic arithmetic operations on natural numbers, such as addition, can be defined by 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 `nat.succ` constructor from the second argument. (Given that addition is symmetric, there is no deep reason for choosing to recurse on the second argument.) Instead of `nat.zero` and `nat.succ n`, Lean also allows us to write `0` and `n + 1` as syntactic sugar.

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

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

Both commands print `9`, as expected. `#eval` employs an optimized interpreter, whereas `#reduce` uses Lean's inference kernel, which is less efficient.

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 `#eval` or `#reduce` calls in your Lean files as documentation.

The definition of multiplication is similar to that of addition:

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

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

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

```
#eval 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
| _ nat.zero     := 1
| m (nat.succ n) := m * power m n
```

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

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

Notice that the recursive call to `power₂` does *not* take `m` as argument. In the entire `def` block introducing `power₂`, `m` is fixed as the first argument. Outside the block, `power₂` is a (curried) binary function, just like `power`. From the outside, there is no difference between the two definitions. In fact, we already saw this syntax for the type argument α of the `list` constructor (Section 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
| nat.zero     := 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
#eval 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 α implicit, 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
#eval append₂ [3, 1] [4, 1, 5]

#check @append₂
#eval @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, propositions, logical formulas, and statements will all mean the same.

In Lean, propositions are simply terms of type `Prop`. This stands in contrast with most descriptions of first-order logic, where terms and formulas are different syntactic entities. A proposition that can be proved is called a theorem (or lemma, corollary, etc.); otherwise it is a nontheorem or false statement. Mathematicians sometimes use the term "proposition" as a synonym for theorem (e.g., "Proposition 3.14"), but in formal logic propositions can also be false.

Here are examples of true statements that can be made about the addition, multiplication, and list reversal operations defined in Section 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 (params₁ : type₁) ... (paramsₘ : typeₘ) :
  statement :=
proof
```

The `:=` symbol separates the lemma's statement and its proof. The syntax of `lemma` is very similar to that of a `def` command without pattern matching, with *`statement`* instead of *`type`* and *`proof`* instead of *`result`*. In the examples above, we put the marker `sorry` as a placeholder for the actual proof. The marker is quite literally an apology to future readers and to Lean for the absence of an actual proof. In Chapters 2 and 3, we will see how to eliminate these `sorry`s.

The intuitive semantic of a `lemma` command with a `sorry` proof is, "Trust me, this proposition should be provable, but I have not carried out the proof yet." Sometimes, we want to express a related idea, namely, "Let us assume this proposition holds from now on." Lean provides the `axiom` command for this, which is often used in conjunction with `constant`(`s`). For example:

```
constants a b : ℤ

axiom a_less_b :
  a < b
```

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

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

Axioms are dangerous, because they rapidly can lead to an inconsistent logic, in which we can derive `false`. For example, if we added a second axiom stating that `a = b`, we could easily derive `b < b`, from which we could derive `false`. The history of interactive theorem proving is paved with inconsistent axioms. An anecdote among many: At the 2020 edition of the Certified Programs and Proofs (CPP) conference, a submitted paper was rejected due to a flawed axiom, from which

one of the peer reviewers derived `false`.[2] Therefore, we will generally avoid axioms, almost always preferring the benign combination of `def` and `lemma` to the dangerous combination of `constant`(`s`) and `axiom`.

Of course, from Lean's point of view, a lemma with a `sorry` proof is effectively an axiom and can be used to play havoc with the logic's consistency. Therefore, it is important to use `sorry` only as a temporary measure, while developing a proof, and not as an alternative to `axiom`.

1.5 Summary of New Constructs

At the end of this and most other chapters, we include a brief summary of the constructs introduced in the chapter. Some syntaxes have multiple meanings, which will be introduced gradually. We refer to *The Lean 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.

Diagnosis Commands

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

Declarations

`axiom`	states an axiom
`constant`(`s`)	declares unspecified new constants or types
`def`	defines a new constant
`inductive`	introduces a type and its constructors
`lemma`	states a lemma and its proof
`namespace` ... `end`	collects declarations in a named scope

Proof Commands

`sorry`	stands for a missing proof or definition

[2] The authors of that submission wrote to report that the axiom has now been revised and derived as a theorem. The entire formal development is now axiom-free. The revised paper has been accepted at the highly competitive conference Computer Aided Verification (CAV) 2020 [8].

Chapter 2

Backward 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 proposition to prove—and either fully proves it or produces new subgoals (or fails). 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].

Tactics are a *backward* proof mechanism. They start from the goal and work backwards towards the already proved lemmas. Consider the lemmas `a`, `a → b`, and `b → c` and the goal `⊢ c`. An informal backward proof is as follow:

> To prove `c`, by `b → c` it suffices to prove `b`.
> To prove `b`, by `a → b` it suffices to prove `a`.
> To prove `a`, we use `a`. □

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

> From `a` and `a → b`, we have `b`.
> From `b` and `b → c`, we have `c`, as desired. □

A forward proof only manipulates theorems, not goals. We will study forward proofs more deeply in Chapter 3.

2.1 Tactic Mode

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

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

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

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

   ```
   ⊢ ∀a b : Prop, a → b → a
   ```

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

   ```
   a b : Prop ⊢ a → b → a
   ```

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

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

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

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

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

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

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

Going back to the Lean proof, we can spare ourselves the `intros` invocations by declaring the variables 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
```

If the proof consists of a single tactic invocation, we can use `by` instead. The syntax `by` *`tactic`* abbreviates `begin` *`tactic`* `end`:

```
lemma fst_of_two_props₃ (a b : Prop) (ha : a) (hb : b) :
  a :=
by apply ha
```

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

```
lemma prop_comp (a b c : Prop) (hab : a → b) (hbc : b → c) :
  a → c :=
begin
  intro ha,
  apply hbc,
  apply hab,
  apply ha
end
```

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

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

2.2 Basic Tactics

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

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

intro(s)

`intro` $[$*name*$]$
`intros` $[$*name*$_1$ ... *name*$_n$$]$

The `intro` tactic moves the leading $\forall$-quantified variable or the leading assumption `a` $\to$ from the goal's target to the local context. The tactic takes as an optional argument the name to give to the variable or to the assumption in the context, overriding the default name. The plural variant `intros` can be used to move several variables or assumptions at once.

refl

The `refl` tactic proves goals of the form $\vdash$ `l = r`, where the two sides `l` and `r` are equal *up to computation*. Computation means expansion 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 global context containing `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 5 = 5 + 5`
ζ-conversion	`(let n : ℕ := 2 in n + n) = 4`
η-conversion	`(λx, f x) = f`
ι-conversion	`prod.fst (a, b) = a`

Since much of Lean's machinery treats terms that are equal up to computation uniformly, it usually makes sense to tell Lean's pretty-printer to aggressively β-reduce its output via the command

```
set_option pp.beta true
```

apply

```
apply lemma-or-hypothesis
```

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

We must invoke `apply` with care, because it can transform a provable goal into an unprovable subgoal. For example, if the goal is `⊢ 2 + 2 = 4` and we `apply` the lemma `false → ?a`, the variable `?a` is matched against `2 + 2 = 4`, and we end up with the unprovable subgoal `⊢ false`. We say that `apply` is *unsafe*. In contrast, `intro` always preserves provability and is therefore *safe*.

exact

```
exact lemma-or-hypothesis
```

The `exact` tactic matches the goal's target with the specified lemma or hypothesis, 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 target and applies it to prove the goal. In the example from Section 2.1, we could have used `assumption` instead of `apply ha`.

cc

The `cc` tactic implements an algorithm known as *congruence closure* [30] to derive new equalities from those that are present in the goal. For example, if the goal contains `b = a` and `f b ≠ f a` as hypotheses, the algorithm will derive `f b = f a` from `b = a` and discover a contradiction with `f b ≠ f a`. The `cc` tactic is also

suitable for more pedestrian work, such as proving `hb : b ⊢ a ∨ b ∨ c` or discovering a contradiction among the goal's hypotheses.

Moreover, `cc` can be used to reason up to associativity (e.g., `(a + b) + c = a + (b + c)`) and commutativity (e.g, `a + b = b + a`). This works for binary operations that are registered as associative and commutative, such as `+` and `*` on arithmetic types, and ∪ and ∩ on sets. We will see an example in Section 2.6.

At this point, you might wonder, "So what does `cc` do exactly?" Of course, you could look up the reference [30] given above to the scientific paper that describes its underlying algorithm, or even read the source code. But this might not be the most efficient use of your time. In truth, even expert users of proof assistants do not fully understand the behavior of the tactics they use daily. The most successful users adopt a relaxed, sporty attitude, trying tactics in sequence and studying the emerging subgoals, if any, to see if they are on the right path.

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

sorry

The `sorry` proof command we encountered in Chapter 1 is also available as a tactic. It "proves" the current goal without actually proving it. Use with care.

2.3 Reasoning about Connectives and Quantifiers

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

```
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. If you want to see the the proof state after the last command, you need to move the cursor right before the `e` of the keyword `end`. Lean simply reports "goals accomplished," meaning that no subgoals remain to be proved.

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

```
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 target against 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., ∧) 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 ∧ to prove the goal `⊢ b ∧ a`, and we apply the two elimination rules to extract `b` and `a` from the hypothesis `a ∧ b`.

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

```
a b : Prop, hab : a ∧ b ⊢ ?m_1 ∧ b
```

The tactic `exact hab` matches `?m_1` (in the target) with `a` (in `hab`). Because variables can occur both in the hypothesis or lemma that is `apply`d and in the target, in general the procedure used to instantiate variables to make two terms syntactically equal is called *unification*. Matching is a special case of unification where one of the two terms contains no variables. In practice, goals with variables are rare, so most of the time Lean's unification amounts to matching.

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

The following is an alternative proof of the lemma `and_swap`:

```
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 ∀-quantified variables instead of parameters of the lemma. Logically, this is equivalent, but in the proof we must then `intro`duce `a` and `b` in addition to `hab`.

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

A third difference is that we now apply, by juxtaposition, `and.elim_right` and `and.elim_left` directly to the hypothesis `a ∧ b` to obtain `b` and `a`, respectively, instead of waiting for the lemmas' assumptions to emerge as new subgoals. This is a small forward step in an otherwise backward proof. The same syntax is used

both to discharge (i.e., prove) a hypothesis and to instantiate a ∀-quantifier. One benefit this approach is that we avoid the potentially confusing `?m_1` variable.

The introduction and elimination rules for disjunction (∨) 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 ∀-quantifiers in `or.intro_left` and `or.intro_right` can be instantiated directly by applying the lemma name to the value we want to instantiate with, via simple juxtaposition. Thus, `or.intro_left false` corresponds to the lemma `?a → ?a ∨ false`. Alternatively, we can invoke `apply or.intro_left` on a goal of the form `... ⊢ c ∨ d`. The lemma is then instantiated with `?a := c` and `b := d`, and the new subgoal is `... ⊢ c`. The first style is forward; the second style, backward.

Both `or.intro_left` and `or.intro_right` are unsafe: If you apply the wrong one of the two, or either of them too early in a proof, you might end up with an unprovable subgoal even if the original goal was provable. This is easy to see if your consider the provable goal `⊢ true ∨ false`: applying `or.intro_right` yields the unprovable subgoal `⊢ false`.

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. You can think of `(?a → ?c) → (?b → ?c) → ?c` as a clever trick to express the concept of disjunction using only implication.

The introduction and elimination rules for equivalence (↔) 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

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

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

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

Again, we instantiate a ∀-quantifier in a forward fashion: `exists.intro 0` is the lemma `?p 0 → (∃x, ?p x)`. The rule is unsafe: Instantiating the quantifier with the wrong term will result in an unprovable goal. For example, if the goal is `⊢ ∃n, n = 1` and we `apply` the lemma `exists.intro 0`, we end up with the unprovable subgoal `⊢ 0 = 1`.

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

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

```
true.intro : true
```

Truth holds no information whatsoever. If it appears as a hypothesis, it is completely useless, and there is no elimination rule that will succeed at extracting any information from it. The `clear` tactic, described in Section 2.8 below, can be used to remove such useless hypotheses.

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

```
false.elim : false → ?a
```

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

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

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

Finally, implication (→) and universal quantification (∀) are the proverbial dogs that did not bark. For both of them, the `intro` tactic is the introduction principle, and application is the elimination principle. For example, given the lemmas `hab : a → b` and `ha : a`, the juxtaposition `hab ha` is a lemma stating `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 target is an implication `P → Q`, invoke `intro hP` to move `P` into your hypotheses: `..., hP : P ⊢ Q`.
- If the goal's target is a universal quantification `∀x : σ, Q`, invoke `intro x` to move `x` into the local context: `..., x : σ ⊢ Q`.
- Look for a lemma or hypothesis whose conclusion has the same shape as the goal's target (possibly containing variables that can be matched), and `apply` it. For example, if the goal's target is `Q` and you have a lemma or hypothesis of the form `hPQ : P → Q`, try `apply hPQ`.
- A negated goal `⊢ ¬ P` is equal to `⊢ P → false` up to computation, so you can invoke `intro hP` to produce the subgoal `hP : P ⊢ false`. Expanding negation's definition by invoking `rewrite not_def` (described in Section 2.5) 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 lemma or hypothesis of the form `P → false` or `¬ P`.
- 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 suspect that you might have reached a dead end, it can make sense to check whether the goal actually is provable under the given assumptions. Even if you started with a provable lemma statement, the current goal might be unprovable (e.g., if you used unsafe rules such as `or.intro_left`).

2.4 Reasoning about Equality

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

```
 eq.refl : ∀a, a = a
 eq.symm : ?a = ?b → ?b = ?a
eq.trans : ?a = ?b → ?b = ?c → ?a = ?c
eq.subst : ?a = ?b → ?p ?a → ?p ?b
```

The first three lemmas are introduction rules specifying that = is an equivalence relation. The fourth lemma is an elimination rule that allows us to replace equals for equals in an arbitrary context, represented by the higher-order variable `?p`.

An example will show how this works. In the proof below, we apply `eq.subst` to rewrite `f a b` to `f a' b`, using the equation `a = a'`:

```
lemma cong_fst_arg {α : Type} (a a' b : α)
    (f : α → α → α) (ha : a = a') :
  f a b = f a' b :=
begin
  apply eq.subst ha,
  apply eq.refl
end
```

The `eq.subst` instance we use has `?a := a`, `?b := a'`, and `?p := (λx, f x b)`:

```
a = a' → (λx, f a b = f x b) a → (λx, f a b = f x b) a'
```

In β-reduced form:

```
a = a' → f a b = f a b → f a b = f a' b
```

The lemma's first assumption is identical to the hypothesis `ha : a = a'`, which we pass as argument in the first `apply` invocation. The lemma's second assumption is a trivial equality that can be proved by `apply eq.refl` or `refl`. The lemma's conclusion matches the goal's target. Notice how a higher-order variable (e.g., `?p`) can represent an arbitrary context (e.g., `f ... b`) around a term (e.g., `a` or `a'`). This works because `apply` reasons up to computation, including β-conversion.

The `eq.subst` lemma can be applied several times in sequence, as follows:

```
lemma cong_two_args {α : Type} (a a' b b' : α)
    (f : α → α → α) (ha : a = a') (hb : b = b') :
  f a b = f a' b' :=
begin
  apply eq.subst ha,
  apply eq.subst hb,
  apply eq.refl
end
```

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

```
lemma cong_two_args₂ {α : Type} (a a' b b' : α)
    (f : α → α → α) (ha : a = a') (hb : b = b') :
  f a b = f a' b' :=
begin
  rewrite ha,
  rewrite hb
end
```

Finally, a note on parsing: Equality binds more tightly than the logical connectives. Thus, `a = a' ∧ b = b'` should be read as `(a = a') ∧ (b = b')`.

2.5 Rewriting Tactics

The rewriting tactic `rewrite` and its relative `simp` replace equals for equals. Unlike `cc`, they use equations as left-to-right rewrite rules. By default, they operate on the goal's target, but they can also be used to rewrite hypotheses specified using the `at` keyword:

`at ⊢`	applies to the target (the default)
`at` h_1 `...` h_n	applies to the specified hypotheses
`at *`	applies to all hypotheses and the target

rewrite

`rewrite` *lemma-or-constant* [`at` *position*]

The `rewrite` tactic applies a single equation as a left-to-right rewrite rule. It searches for the first subterm that matches the rule's left-hand side; once found, all occurrences of that subterm are replaced by the right-hand side of the rule. If the rule contains variables, these are instantiated as necessary. To apply a lemma as a right-to-left rewrite rule, put a short left arrow (←) in front of its name.

Given the lemma `l : ∀x, g x = f x` and the goal `⊢ h (f a) (g b) (g c)`, the tactic `rewrite l` produces the subgoal `⊢ h (f a) (f b) (g c)`, whereas `rewrite ←l` produces the subgoal `⊢ h (g a) (g b) (g c)`.

Instead of a lemma, we can also specify the name of a constant. This will attempt to use one of the constant's defining equations as rewrite rules. Exceptionally, this does not work with `not` (¬). We must use `rewrite not_def` instead.

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. Unlike `rewrite`, `simp` can rewrite terms containing bound variables (e.g., occurrences of `x` in the body of `λx, ...`, `∀x, ...`, or `∃x, ...`).

`simp [`*lemma-or-constant*$_1$`, ..., `*lemma-or-constant*$_n$`]` [`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 hypotheses.

The minus sign (-) in front of a lemma name temporarily removes the lemma from the simp set. A powerful incantation that both simplifies the hypotheses and uses the result to simplify the goal's target is `simp [*] at *`.

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

To find out what `simp` does, you can enable tracing via the command

```
set_option trace.simplify.rewrite true
```

2.6 Proofs by Mathematical Induction

Lean's `induction` tactic performs structural induction on an inductive type. *Structural induction* simply means that the induction follows the structure of the inductive type. For natural numbers constructed from `nat.zero` and `nat.succ`, structural induction corresponds to standard mathematical induction: To prove `p n`, it suffices to prove `p 0` and `∀k, p k → p (k + 1)`. Equipped with `induction`, we can reason about the addition and multiplication operations we defined by recursion in Section 1.3.

Addition is defined by recursion on its second argument. We will prove two lemmas, `add_zero` and `add_succ`, that give us alternative equations that recurse 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 first `{ }` block corresponds to the base case; the second block corresponds to the induction step. The name `n_ih`, in the induction step, is generated by the `induction` tactic.

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,
  { refl },
  { simp [add, n_ih] }
end

lemma add_comm (m n : ℕ) :
  add m n = add n m :=
begin
  induction n,
  { simp [add, add_zero] },
```

```
  { simp [add, add_succ, n_ih] }
end

lemma add_assoc (l m n : ℕ) :
  add (add l m) n = add l (add m n) :=
begin
  induction n,
  { refl },
  { simp [add, n_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 `cc`, know about this. Here is the syntax to achieve this for `add`:

```
@[instance] def add.is_commutative : is_commutative ℕ add :=
{ comm := add_comm }

@[instance] def add.is_associative : is_associative ℕ add :=
{ assoc := add_assoc }
```

(The `@[instance]` mechanism will be explained in Chapter 4.) The following example uses the `cc` tactic to reason up to associativity and commutativity of `add`:

```
lemma mul_add (l m n : ℕ) :
  mul l (add m n) = add (mul l m) (mul l n) :=
begin
  induction n,
  { refl },
  { simp [add, mul, n_ih],
    cc }
end
```

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

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

2.7 Induction Tactic

induction

`induction` *term* [`with` $name_1 \ldots name_n$]

The `induction` tactic performs structural induction on the specified term. This gives rise to as many subgoals as there are constructors in the definition of the term's type. Induction hypotheses are available as hypotheses in the subgoals corresponding to recursive constructors (e.g., `nat.succ` or `list.cons`). The optional names $name_1, \ldots, name_n$ are used for any emerging variables or hypotheses.

2.8 Cleanup Tactics

The following tactics help us clean up the goal by allowing us to give more meaningful names to variables or hypotheses or to remove useless hypotheses. We did not need them so far, but they can be helpful during proof exploration.

rename

`rename` *`variable-or-hypothesis new-name`*

The `rename` tactic renames a variable or hypothesis.

clear

`clear` $\mathit{variable\text{-}or\text{-}hypothesis}_1 \ldots \mathit{variable\text{-}or\text{-}hypothesis}_n$

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

2.9 Summary of New Constructs

Commands

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

Attribute

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

Proof Commands

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

Tactics

`apply`	matches the goal's target with the lemma's conclusion
`assumption`	proves the goal using a hypothesis
`cc`	propagates equalities up to to associativity and commutativity
`clear`	removes a variable or hypothesis from the goal
`exact`	proves the goal using the specified lemma
`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 variable or hypothesis
`rewrite`	applies the given rewrite rule once
`simp`	applies a set of preregistered rewrite rules exhaustively
`sorry`	stands for a missing proof or definition

Tactic Combinator

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

Chapter 3

Forward Proofs

Tactics are a backward 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. Most users combine the two styles, using whichever seems the most appropriate for the situation. The higher readability of structured proofs make them popular with some users, especially in mathematical circles.

Structured proofs are syntactic sugar sprinkled over Lean's *proof terms*. They are built using keywords such as `assume`, `have`, and `show` that mimic pen-and-paper proofs. All Lean proofs, whether tactical or structured, are reduced internally to proof terms. We have seen some specimens already, in Chapter 2: Given hypotheses `ha : a` and `hab : a → b`, the term `hab ha` is a proof term for the proposition `b`, and we write `hab ha : b`. The names of lemmas and hypotheses, such as `ha` and `hab`, are also proof terms. Pushing this idea further, given `hbc : b → c`, we can build the proof term `hbc (hab ha)` for the proposition `c`. We can think of `hab` as a function that converts a proof of `a` to a proof of `b`, and similarly for `hbc`.

Structured proofs are the default in Lean. They can be used outside tactic mode. To enter tactic mode, we need to use a `begin ... end` block or a `by` line.

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 [23] and Van Raamsdonk's lecture notes [28] are other useful references.

3.1 Structured Proofs

As a first example, consider the following structured proof:

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

Each variable bound by a ∀-quantifier and each assumption of an implication is introduced explicitly in the proof using the `fix` and `assume` commands. Several

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

Informally, we could write the proof as follows:

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

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

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

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

Informally:

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

Notice that this is a forward proof: It progresses one theorem at a time from the hypothesis `a` to the desired theorem `c`.

In general, the `fix–assume–show` skeleton simply repeats the lemma's conclusion. In between, as many `have` commands as desired may appear, depending on how detailed we want the argument to be. Details can increase readability, but providing too many details can overwhelm even the most persistent the reader.

The `have` command has a similar syntax to `lemma` but appears inside a structured proof. We can think of a `have` as a definition. In `have hb : b := hab ha`, the right-hand side `hab ha` is a proof term for `b`, and the left-hand side `hb` is defined as a synonym for that proof term. From that point on, `hb` and `hab ha` can be used

interchangeably. Since `hb` and `hc` are used only once and their proofs are very short, experts would tend to inline their proofs, replacing `hc` by `hbc hb` and `hb` by `hab ha`, yielding

```
lemma prop_comp₂ (a b c : Prop) (hab : a → b) (hbc : b → c) :
  a → c :=
assume ha : a,
show c, from
  hbc (hab ha)
```

A typical structured proof has the following format:

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

3.2 Structured Constructs

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

Lemma or Hypothesis

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

```
lemma two_add_two_eq_four :
  2 + 2 = 4 :=
begin
  ...
end
```

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

```
lemma this_time_with_feelings :
  2 + 2 = 4 :=
two_add_two_eq_four
```

We can pass arguments to lemmas to instantiate ∀-quantifiers and discharge assumptions. Suppose the lemma `add_comm (m n : ℕ) : add m n = add n m` is available, and suppose we want to prove its instance `add 0 n = add n 0`. This can be achieved neatly using the name of the 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. The `exact` tactic can be seen as the inverse of `by`. Why enter tactic mode if only to leave it immediately?

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

fix

```
fix names [: type],
fix (names₁ [: type₁]) ... (namesₙ [: typeₙ]),
```

The `fix` command moves ∀-quantified variables from the goal's target to the local context. It can be seen as a structured version of the `intros` tactic.

assume

```
assume names [: proposition],
assume (names₁ [: proposition₁]) ... (namesₙ [: typeₙ]),
```

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

have

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

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

let

```
let name [: type] := term 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 is designed for computable data, not proofs. Expanding or introducing a `let` corresponds to ζ-conversion (Section 2.2).

show

```
show proposition, from
  proof
```

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

3.3 Forward Reasoning about Connectives and Quantifiers

Reasoning about the logical connectives and quantifiers in a forward fashion uses the same introduction and elimination rules as in tactic mode (Section 2.3). A few examples will show the flavor. Let us start with conjunction:

```
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
```

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

```
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
```

On the other hand, the tactical proof is arguably easier to derive, in a mindless fashion. In general, backward proofs are easier to find, and most of the automation found in proof assistants works backwards. The reason is simple: Pretend that you are Miss Marple or Hercule Poirot on a murder investigation. A backward investigation would start from the crime scene and try to extract clues that potentially connect a handful of suspects to the crime. In contrast, a forward investigation might start by questioning as many as seven billion people to determine whether they have an alibi. Which is more likely to succeed?

Our next examples concern *one-point rules*. These are lemmas that can be used to eliminate a quantifier when the bound variable can effectively take only one value. For example, the proposition `∀n, n = 666 → beast ≥ n` can be simplified to `beast ≥ 666`. The next lemma justifies this simplification:

```
lemma forall.one_point {α : Type} (t : α) (p : α → Prop) :
  (∀x, x = t → p x) ↔ p t :=
iff.intro
  (assume hall : ∀x, x = t → p x,
   show p t, from
     begin
       apply hall t,
       refl
     end)
  (assume hp : p t,
   fix x,
   assume heq : x = t,
   show p x, from
     begin
       rewrite heq,
       exact hp
     end)
```

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

```
iff.intro (_) (_)
```

The two placeholders are important to make this proof well formed. We already put parentheses because we strongly suspects that these will be nontrivial proofs. Because proofs are basically terms, which are basically programs, the advice we gave in Section 1.1.4 applies here as well:

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

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

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

Each of the remaining placeholder can be replaced by a structured proof or by a tactical proof. To fill these placeholders, we can use essentially the same procedure as for exhibiting an inhabitant of a type (Section 1.1.4), interpreting implication as the function arrow and writing `assume`–`show` instead of λ.

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

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

It works. Matching `forall.one_point t p` against the statement of `beast_666` yields the instantiation `t := 666` and `p := (λm, beast = m)`. Indeed, if we substitute these values in `forall.one_point` and β-reduce, we get the statement of `beast_666`.

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

```
lemma exists.one_point {α : Type} (t : α) (p : α → Prop) :
  (∃x : α, x = t ∧ p x) ↔ p t :=
iff.intro
  (assume hex : ∃x, x = t ∧ p x,
   show p t, from
     exists.elim hex
       (fix x,
        assume hand : x = t ∧ p x,
        show p t, from
          by cc))
  (assume hp : p t,
   show ∃x : α, x = t ∧ p x, from
     exists.intro t
       (show t = t ∧ p t, from
          by cc))
```

Notice how we use `exists.elim hex` to obtain an `x` such that `x = t ∧ p x`. This admittedly is a bit cumbersome.

3.4 Calculational Proofs

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

The general syntax is as follows:

```
calc    term₀
     op₁ term₁ :
  proof₁
... op₂ term₂ :
  proof₂
     ⋮
```

```
... op_n term_n :
  proof_n
```

The horizontal dots (`...`) are part of the syntax. Each $proof_i$ justifies the statement $term_{i-1}\ op_i\ term_i$. The operators op_i need not be identical, but they must be compatible. 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 rewrite two_mul
... = m + n + m :
  by cc
```

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

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

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

```
lemma two_mul_example₂ (m n : ℕ) :
  2 * m + n = m + n + m :=
have h₁ : 2 * m + n = (m + m) + n :=
  by rewrite two_mul,
have h₂ : (m + m) + n = m + n + m :=
  by cc,
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.

3.5 Forward Tactics

The structured proof commands `have` and `let` are are also available as tactics. Many users, especially beginners, prefer the tactic mode. Even in tactic mode, it can be useful to reason in a forward fashion.

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

```
lemma prop_comp₃ (a b c : Prop) (hab : a → b) (hbc : b → c) :
  a → c :=
begin
  intro ha,
```

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

have

```
have [[name :] proposition] :=
  proof
```

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

let

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

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

Observe that the syntax for the `let` tactic is slightly different than for the `let` structured proof command, with no `in` keyword (but a comma instead if the tactic is followed by another tactic).

3.6 Dependent Types

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

Consider a function `pick` that takes a natural number `n`—i.e, a value from $\mathbb{N}$ `= {0, 1, 2, ...}`—and that returns a natural number between `0` and `n`. Intuitively, `pick n` should have the type `{0, 1, ..., n}`—i.e., the type consisting of all natural numbers `i` that $\texttt{i} \leq \texttt{x}$. In Lean, this is written $\texttt{\{i : } \mathbb{N} \texttt{ // i} \leq \texttt{ n\}}$. This would be the type of `pick n`. Mathematically inclined readers may want to think of `pick` as an $\mathbb{N}$-indexed family of constants

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

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

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

This is a dependent type: The type of the result depends on the value of the argument `n`. The name of the argument `n` is immaterial.

Unless otherwise specified, a *dependent type* means a type depending on a (non-type) term, as above, with `n : ℕ` as the term and `{i : ℕ // i ≤ n}` 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 `λα : Type, list α`, or the polymorphic type `λα : Type, α → α` of functions with the same domain and codomain. A term may depend on a type—for example, the polymorphic identity function `λα : Type, λx : α, x`. And of course, a term may also depend on a term—for example, `λn : ℕ, n + 2`. In summary, there are four cases for `λx, t`:

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

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

The App and Lam rules presented in Section 1.1.3 must be generalized to work with dependent types:

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

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

The notation `τ[x]` stands for a type that may contain `x`, and `τ[u]` stands for the same type where all occurrences of `x` have been replaced by `u`.

The simply typed case arises when `x` does not occur in `τ[x]`. Then, we can simply write `σ →` instead of `(x : σ) →`. The familiar notation `σ → τ` is equivalent to `(_ : σ) → τ`. It is easy to check that App′ and Lam′ coincide with App and Lam when `x` does not occur in `τ[x]`. The example below demonstrates App′:

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

The next example demonstrates Lam′:

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

The picture is incomplete because we only check that the terms—the entities on the left-hand side of a colon (`:`)—are well typed. The types—the entities on the right-hand side of a colon—must also be checked, using the same type system, but this time as terms. For example, the type of `nat.succ` is `ℕ → ℕ`, whose type

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

is `Type`. Types of types, such as `Type` and `Prop`, are called universes. We will study them more closely in Chapter 11.

Regrettably, the intuitive syntax `(x : σ) → τ`, which we have used above, is not available in Lean. Instead, we must write `∀x : σ, τ` to specify a dependent type. The familiar notation `σ → τ` is supported as syntactic sugar for `∀_ : σ, τ`. The `∀` syntax emphasizes that `x` is a bound variable. As an alias for `∀`, we can also write `Π`. 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: `list.nil` is a function that takes a type `α` as argument and that returns a value of type `list α`.

3.7 The Curry–Howard Correspondence

You have likely noticed that the same symbol `→` is used both for implication (e.g., `false → true`) and as the type constructor of functions (e.g., `ℤ → ℕ`). Similarly, `∀` is used both as a quantifier and to specify dependent types. Without context, we cannot tell whether `a → b` refers to the type of a function with domain `a` and codomain `b` or to the proposition "`a` implies `b`," and similarly, `∀x : a, b[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*(*–De Bruijn*) *correspondence*. It is also called the *PAT principle*, where PAT is a double mnemonic:

PAT = propositions as types PAT = proofs as terms

Furthermore, because types are also terms, we also have that propositions are terms. However, PAT is emphatically not a quadruple mnemonic:

~~PAT = proofs as types~~

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. The question "Is `H` a proof of `P`?" becomes equivalent to "Does the term `H` have type `P`?" As a result, inside of Lean, there is no proof checker, only a type checker.

Let us go through the dramatis personae one by one. We use the metavariables `σ`, `τ` for types; `P`, `Q` for propositions; `t`, `u`, `x` for terms; and `h`, `G`, `H`, for proofs. Starting with "propositions as types," for types, we have the following:

1. `σ → τ` is the type of (total) functions from `σ` to `τ`.
2. `∀x : σ, τ[x]` is the dependent function type from `x : σ` to `τ[x]`.

In contrast, for propositions, we have the following:

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

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

1. A constant is a term.
2. A variable is a term.
3. `t u` is the application of function `t` to argument `u`.
4. `λx, t[x]` is a function mapping `x` to `t[x]`.

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

1. The name of a lemma or hypothesis is a proof.
2. `H t`, which instantiates the leading parameter or quantifier of proof `H`'s statement with term `t`, is a proof.
3. `H G`, which discharges the leading assumption of `H`'s statement with proof `G`, is a proof. This operation is called *modus ponens*.
4. `λh : P, H[h]` is a proof of `P → Q`, assuming `H[h]` is a proof of `Q` for `h : P`.
5. `λx : σ, H[x]` is a proof of `∀x : σ, Q[x]`, assuming `H[x]` is a proof of `Q[x]` for `x : σ`.

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

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

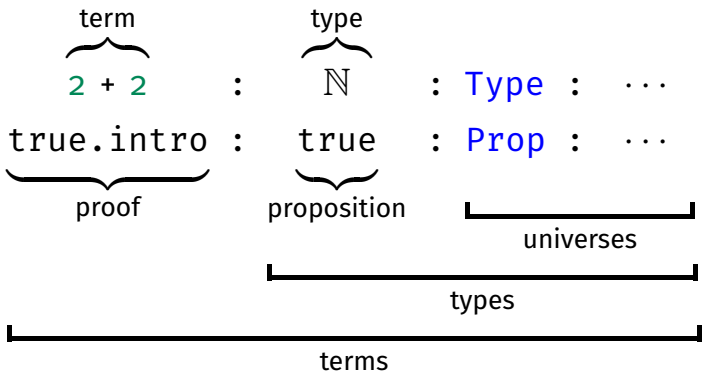
```
lemma case_5 :
  ∀x : σ, Q[x] :=
fix x,
show Q[x], from
  H[x]
```

Some commands are provided in Lean under two different names but are really the same as per the Curry–Howard correspondence. This is the case for `constant` and `axiom` as well as for `fix` and `assume`. There are also pairs with slightly different behavior. This is the case for `def` and `lemma` as well as for `let` and `have`. The fundamental difference is this: When we define some function or data, we care not only about the type but also about the body—the behavior. On the other hand, once we have proved a lemma, the proof becomes *irrelevant*. All that matters is that there *is* a proof. We will return to the topic of proof irrelevance in Chapter 11.

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

Tactical proof	Structured proof	Raw proof term
`intro x,`	`fix x,`	`λx,`
`intro h,`	`assume h,`	`λh,`
`have k := _,`	`have k := _,`	`have k := _,`
`let x := _,`	`let x := _ in`	`let x := _ in`
`exact (_ : P)`	`show P, from _`	`_ : P`

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



According to the broad senses, any expression is a term, any expression that may occur on the right-hand side of a typing judgment is a type, and any expression that may occur on the right-hand side of a typing judgment with a type on its left-hand side is a universe. This is perfectly consistent with the reading of `t : u` as "`t` has type `u`" and the notion that universes are types of types.

3.8 Induction by Pattern Matching

In Section 2.6, we saw how to use the `induction` tactic to perform proofs by induction. An alternative, more flexible style relies on pattern matching and the Curry–Howard correspondence.

Recall the definition of the reverse of a list from Section 1.3:

```
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 useful property to prove is that `reverse` is its own inverse: `reverse (reverse xs) = xs` for all lists `xs`. However, if we try to prove it by induction, we quickly run into an obstacle. The induction step is

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

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

In the proof of the lemma, the patterns on the left, `[]` and `x :: xs`, correspond to the two constructors of the ∀-quantified variable `xs :: list α`. On the right-hand side of each `:=` symbol is a proof for the corresponding case.

Inside the induction step's proof, the induction hypothesis is available under the same name as the lemma we are proving (`reverse_append`). We explicitly pass `xs` as argument to the induction hypothesis. This is useful as documentation, and it also prevents Lean from accidentally invoking the induction hypothesis in a circular fashion on `x :: xs`. Lean's termination checker would notice that the argument is ill-founded and raise an error.

For reference, the corresponding tactical proof is as follows:

```
lemma reverse_append₂ {α : Type} (xs ys : list α) :
  reverse (xs ++ ys) = reverse ys ++ reverse xs :=
begin
  induction xs,
  case list.nil {
    simp [reverse] },
  case list.cons : x xs ih {
    simp [reverse, ih] }
end
```

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

Simultaneous pattern matching on multiple variables is supported (e.g., `xs` and `ys` above). The patterns are then separated by spaces. This is the reason why parentheses are necessary around complex patterns. The general format is

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

Notice the strong similarity with the syntax of `def` (Section 1.3). The two commands are, in fact, almost the same, the main exception being that `lemma` considers the defined term or the proof opaque, whereas `def` keeps it transparent. Since the actual proofs are irrelevant once a lemma is proven (Section 11), there is no need to expand them later. A similar distinction exists between `let` and `have`.

By the Curry–Howard correspondence, a proof by induction by pattern matching is the same as a recursive proof term. When we invoke the induction hypothesis, we are really just invoking a recursive function recursively. This explains why the induction hypothesis has the same name as the lemma we prove. Lean's termination checker is used to establish well-foundedness of the proof by induction.

With the `reverse_append` lemma in place, we can prove our initial goal:

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

```
  | (x :: xs) :=
    by simp [reverse, reverse_append, reverse_reverse xs]
```

Induction by pattern matching is highly popular among Lean users. Its main advantages are its convenient syntax and its support for well-founded induction and not only structural induction, as provided by the `induction` tactic (Section 2.7). However, in this guide, we will not need the full power of well-founded induction. Furthermore, for subtle logical reasons, induction by pattern matching is not available for inductive predicates, which are the topic of Chapter 5.

3.9 Summary of New Constructs

Proof Commands

`assume`	states assumptions
`calc`	combines proofs by transitivity
`fix`	fixes variables
`have`	states an intermediate lemma
`let ... in`	introduces a local definition
`show ..., from`	states the target

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 essence of typed functional programming: inductive types, proofs by induction, recursive functions, pattern matching, structures (records), and type classes. The concepts covered here are described in more detail in Chapters 7 to 9 of *Theorem Proving in Lean* [1].

4.1 Inductive Types

Inductive types are modeled after the data types of typed functional programming languages (e.g., Haskell, ML, OCaml). 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 binary trees. We also take a brief look at vectors of length *n*, a dependent type.

Recall the definition of natural numbers as an inductive type:

```
inductive nat : Type
| zero : nat
| succ : nat → nat
```

This definition introduces three constants, which could have been declared as follows:

```
constant nat : Type
constant nat.zero : nat
constant nat.succ : nat → nat
```

However, the inductive definitions also asserts some properties about `nat.zero` and `nat.succ`, which is why we use the `inductive` command. It also introduces further constants that are used internally to support induction and recursion.

As we saw in Section 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, ε, ∞, or `NaN` that cannot be expressed using a finite combination of `nat.zero`

and `nat.succ`,[1] and "no confusion" ensures that `nat.zero` $\neq$ `nat.succ n` for all `n` and that `nat.succ` is injective. In addition, values of inductive types are always finite. The infinite term

```
nat.succ (nat.succ (nat.succ (nat.succ ...)))
```

is not a value. Nor does there exist a number `n` such that `nat.succ n = n`, as we will formally prove below.

Inductive types are very convenient to use, because they support induction and recursion and their constructors are well behaved, but not all types can be defined as an inductive type. In particular, numeric types such as $\mathbb{Q}$ (the rationals) and $\mathbb{R}$ (the real numbers) require more elaborate constructions, based on quotienting and subtyping. These will be explained in Chapters 11 and 13.

4.2 Structural Induction

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

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

We can of course also write `nat.zero` and `nat.succ k`.

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

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

Structural induction on `n` then produces the two subgoals

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

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

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

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

[1] We could try to cheat and axiomatize special constants using `constant` and `axiom`, but the result would be either tautological (i.e., useless) or contradictory (i.e., positively harmful). A safer but equally pointless alternative would be to locally assume the existence of such a value; this would be pointless because the assumption could never be discharged.

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

For arithmetic expressions, the base cases are

```
i : ℤ ⊢ P[aexp.num i]          x : string ⊢ P[aexp.var x]
```

and the induction steps are

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

Notice the two induction hypotheses: one about e_1 and one about e_2.

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

Regardless of the inductive type τ, the procedure to compute the subgoals is always the same:

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

As an example, we will prove that `nat.succ n` $\neq$ `n` for all `n` : $\mathbb{N}$. We start with an informal proof, because these require us to understand what we are doing:

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

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

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

Now let us carry out the proof in Lean:

```
lemma nat.succ_neq_self (n : ℕ) :
  nat.succ n ≠ n :=
```

```
begin
  induction n,
  { simp },
  { simp [n_ih] }
end
```

The routine reasoning about constructors is all carried out by `simp` automatically, which is usually what we want.

We can supply our own names, and reorder the cases, by using the `case` tactic in front of each case, together with the case's name and the desired names for the variables and hypotheses introduced by `induction`. For example:

```
lemma nat.succ_neq_self₂ (n : ℕ) :
  nat.succ n ≠ n :=
begin
  induction n,
  case nat.succ : m ih {
    simp [ih] },
  case nat.zero {
    simp }
end
```

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

4.3 Structural Recursion

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

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

The constructor we peel off here is `nat.succ` (written `+ 1`). Such functions are guaranteed to call themselves only finitely many times before the recursion stops; for example, `fact 12345` will call itself 12345 times. This is a prerequisite for establishing that the function terminates, an important property to ensure both logical consistency and termination of evaluation.

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

```
def fact₂ : ℕ → ℕ
| 0       := 0
| 1       := 1
| (n + 1) := (n + 1) * fact₂ n
```

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

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

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

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

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

```
constant immoral : ℕ → ℕ

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

lemma proof_of_false :
  false :=
have immoral 0 = immoral 0 + 1 :=
  immoral_eq 0,
have immoral 0 - immoral 0 = immoral 0 + 1 - immoral 0 :=
  by cc,
have 0 = 1 :=
  by simp [*] at *,
show false, from
  by cc
```

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

The observant reader will have noticed that the above definitions of factorial are mathematically wrong: `fact` shockingly return `0` regardless of the argument, and $\texttt{fact}_2\ \texttt{0}$ should give `1`, not `0`. We quite literally `fact`ed up. These embarrassing mistakes remind us to *test* our definitions and *prove* some properties about them. Although flawed axioms arise now and then, what is much more common are definitions that fail to capture the intended concepts. Just because a function is called `fact` does not mean that it actually computes factorials.

4.4 Pattern Matching Expressions

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

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

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

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

```
def bcount {α : Type} (p : α → bool) : list α → ℕ
| []        := 0
| (x :: xs) :=
  match p x with
  | 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`. As a general rule, we will use type `bool`, of Booleans, within programs and use the type `Prop`, of propositions, when stating properties of programs. The two values of type `bool` are called `tt` and `ff`.[2] The connectives are called `bor` (infix: `||`), `band` (infix: `&&`), `bnot`, and `bxor`.

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

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

This works only for propositions that are decidable—i.e., executable. This is the case for $\leq$: Given concrete values for the arguments, such as 35 and 49, Lean can reduce $35 \leq 49$ to `true`. Lean keeps track of decidability using a mechanism called type classes, which will be explained below.

4.5 Structures

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

The definition below introduces a structure called `rgb` with three fields of type $\mathbb{N}$ called `red`, `green`, and `blue`:

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

[2] Lean also allows us to use `true` and `false`, but these are then implicitly converted from `Prop` to `bool`. We generally recommend to avoid relying on such implicit coercions.

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

```
inductive rgb : Type
| mk : ℕ → ℕ → ℕ → rgb

def rgb.red : rgb → ℕ
| (rgb.mk r _ _) := r

def rgb.green : rgb → ℕ
| (rgb.mk _ g _) := g

def rgb.blue : rgb → ℕ
| (rgb.mk _ _ b) := b
```

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

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

The general syntax to define structures is

```
structure structure-name (params₁ : type₁) ... (paramsₖ : typeₖ)
  [extends structure₁, ..., structureₘ] : Type :=
(field-names₁ : field-type₁)
   ⋮
(field-namesₙ : field-typeₙ)
```

The parameters $params_1, \ldots, params_k$ can be seen as additional fields, but unlike $field\text{-}names_1, \ldots, field\text{-}names_n$, they are also stored in the type, as arguments to the type constructor (*structure-name*).

Values can be specified in a variety of syntaxes:

```
def pure_red : rgb :=
{ red   := 0xff,
  green := 0x00,
  blue  := 0x00 }

def semitransparent_red : rgba :=
{ alpha := 0x7f,
  ..pure_red }
```

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

Next, we define an operation called `shuffle`:

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

The definition relies on the generated selectors `rgb.red`, `rgb.green`, and `rgb.blue`. Instead of `rgb.red c`, we could also write `c.red`, and similarly for the other

fields. Sometimes we will see these forms in Lean's output, even if we do not use them ourselves.

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 :=
begin
  cases c,
  refl
end
```

The proof teaches us a new trick. The `cases` tactic is a close relative of `induction`. It performs a case distinction on its argument, but it does not generate induction hypotheses. For `rgb`, the `cases c` tactic transforms a goal of the form `⊢ P[c]` into a subgoal `r g b : ℕ ⊢ P[rgb.mk r g b]`. We could also have written `induction c`.

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

```
lemma shuffle_shuffle_shuffle₂ (c : rgb) :
  shuffle (shuffle (shuffle c)) = c :=
match c with
| rgb.mk r g b := eq.refl _
end
```

Recall from Section 2.4 that `eq.refl` is the property `∀a, a = a`.

4.6 Type Classes

Type classes are a mechanism that was popularized by Haskell and that is present in several proof assistants. In Lean, a *type class* is a structure type combining abstract constants and their properties. A type can be declared an instance of a type class by providing concrete definitions for the constants and proving that the properties hold. Based on the type, Lean retrieves the relevant instance.[3]

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

```
@[class] structure inhabited (α : Type) : Type :=
(default : α)
```

The syntax is the same as for structures, except with the attribute `@[class]` in front. The parameter `α` represents an arbitrary type that could be a member of this class.

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

```
@[instance] def nat.inhabited : inhabited ℕ :=
{ default := 0 }
```

This defines a structure value called `nat.inhabited` of type `inhabited ℕ`. In addition, thanks to the `@[instance]` attribute, the structure value is registered as *the*

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

canonical instance to use whenever a structure of type `inhabited ℕ` is desired. In the global table storing type class instances, there is now an entry `inhabited ℕ ↦ nat.inhabited`.

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

```
@[instance] def list.inhabited {α : Type} :
  inhabited (list α) :=
{ default := [] }
```

This adds the entry `inhabited (list α) ↦ list.inhabited` to the global table.

Sometimes we may want to supply several instances of a given type class for the same type. Lean will then choose the first matching instance it finds. As an example, recall that finite functions of type $\alpha \to \beta$ are isomorphic to their function tables, of type $\beta \times \cdots \times \beta$ (with $|\alpha|$ copies of β). Accordingly, $|\alpha \to \beta| = |\beta|^{|\alpha|}$. To make this 0, we must have both $|\beta| = 0$ and $|\alpha \neq 0|$. In other words, the type $\alpha \to \beta$ is inhabited if either β is inhabited or α is not inhabited. There is no type class for uninhabitedness, but the type `empty` is not inhabited, so it will do:

```
@[instance] def fun.inhabited {α β : Type} [inhabited β] :
  inhabited (α → β) :=
{ default := λa : α, inhabited.default β }

inductive empty : Type

@[instance] def fun_empty.inhabited {β : Type} :
  inhabited (empty → β) :=
{ default := λa : empty, match a with end }
```

The first instance relies itself on an instance of the same type class but on a different type. This frequently occurs and is part of the reason why we speak of type class *search* and not type class lookup. In the second instance, notice the trivial `match` that implements the body of a function that can never be called, because no value of type `empty` can be passed as argument. The definition of `empty` has zero introduction rules, so the corresponding `match` has zero cases.

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

```
@[instance] def prod.inhabited {α β : Type}
    [inhabited α] [inhabited β] :
  inhabited (α × β) :=
{ default := (inhabited.default α, inhabited.default β) }
```

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

```
def head {α : Type} [inhabited α] : list α → α
| []       := inhabited.default α
| (x :: _) := x
```

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

The syntax `[inhabited α]` adds an implicit argument to the `head` constant. 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 running the command

```
#eval head ([] : list ℕ)   -- result: 0
```

Lean will search for an instance for `inhabited ℕ` and will find `nat.inhabited`, the instance we declared above. In that declaration, we set `default` to be `0` and hence this is what `#eval` prints. If multiple instances are applicable and Lean chooses the wrong one, we can use the `@` syntax to transform type class arguments into explicit arguments and supply the desired type class instance. This is the main reason why we give names to the type class instances.

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

```
inhabited.default (α : Type) [i : inhabited α] : α
```

A difference between ordinary structures and type classes is that the selectors of structures use parentheses `( )` whereas those of type classes use square brackets `[ ]`. When we used `inhabited.default α` to define the `head` function, Lean looked for an instance of `inhabited α` in the global table of registered instances and in the local context. The global table contained an entry for `inhabited ℕ` but none that matched `inhabited α`, so this was doomed to fail. On the other hand, the local context contained an anonymous parameter of type `inhabited α`, which could be used.

Lean's core library defines `list.head` exactly as we did. In practice, almost all types are nonempty (with the notable exceptions of `empty` and `false`), so the `inhabited` restriction is hardly an issue. Regardless, lists over empty types are uninteresting—the only possible value is `[]`.

We can prove abstract lemmas about the type class `inhabited`, such as

```
lemma head_head {α : Type} [inhabited α] (xs : list α) :
  head [head xs] = head xs
```

It is important to add the assumption `[inhabited α]` to be allowed to use the operator `head` on lists of type `list α`. If we omit this assumption, Lean will raise an error telling us that type class synthesis failed.

There are more type classes requiring only a constant but no properties, including the following:

```
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 a rich hierarchy of algebraic type classes (groups, monoids, ring, fields, etc.) and to allow overloading of common symbols such as `+`, `*`, `0`, `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 contain properties that restrict how the given constants behave.

In Section 2.6, we encountered the `is_commutative` and `is_associative` type classes:

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

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

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

Conceptually, `is_commutative` is a dependent type of triples `(α, f, comm)`, and similarly for `is_associative`. The type of `f` depends on `α`, and the type of `comm` depends on `α` and `f`. Although they are parameters, `α` and `f` are also stored along with `comm`. But the type class mechanism treats the parameters differently from the fields: The parameters are treated as *input* to the type class search, whereas the fields are *output*.

In Section 2.6, we registered our `add` function on $\mathbb{N}$ as a commutative and associative operation:

```
@[instance] def add.is_commutative : is_commutative ℕ add :=
{ comm := add_comm }

@[instance] def add.is_associative : is_associative ℕ add :=
{ assoc := add_assoc }
```

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

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

```
@[class] structure class-name (params₁ : type₁) ... (paramsₖ : typeₖ)
  [extends structure₁, ..., structureₘ] :=
(constant-names₁ : constant-type₁)
   ⋮
(constant-namesₙ : constant-typeₙ)
(property-names₁ : proposition₁)
```

```
    ⋮
(property-names_p : proposition_p)
```

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

```
@[instance] def instance-name : type-class arguments :=
{ constant_1 := definition_1,
    ⋮
  constant_n := definition_n,
  property_1 := proof_1,
    ⋮
  property_p := proof_p }
```

4.7 Lists

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

In the first example, we 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`:

```
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,
  clear h,
  cc
end
```

As a result of the case distinction, `y` is replaced by `x` and `ys` is replaced by `xs` throughout the goal, yielding the subgoal

```
h : x :: xs = x :: xs ⊢ x = x ∧ xs = xs
```

The `clear h` tactic removes the useless hypothesis `h : x :: xs = x :: xs`.

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
```

The first 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 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
```

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

```
lemma map_ident {α : Type} (xs : list α) :
  map (λx, x) xs = xs :=
begin
  induction xs,
  case list.nil {
    refl },
  case list.cons : y ys ih {
    simp [map, ih] }
end
```

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 g (map f xs) = map (λx, g (f x)) xs :=
begin
  induction xs,
  case list.nil {
    refl },
  case list.cons : y ys ih {
    simp [map, ih] }
end
```

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

```
lemma map_append {α β : Type} (f : α → β) (xs ys : list α) :
  map f (xs ++ ys) = map f xs ++ map f ys :=
begin
  induction xs,
  case list.nil {
    refl },
  case list.cons : y ys ih {
    simp [map, ih] }
end
```

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

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

```
def tail {α : Type} : list α → list α
| []       := []
| (_ :: xs) := xs
```

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

The counterpart of `tail` is a function that extracts the first element of a list. We already reviewed one solution in Section 4.6. Another possible definition uses an `option` wrapper:

```
def head_opt {α : Type} : list α → option α
| []       := option.none
| (x :: _) := option.some x
```

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

```
match head_opt xs with
| option.none   := handle_the_error
| option.some x := do_something_with_value x
end
```

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

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

```
def head_le {α : Type} : ∀xs : list α, xs ≠ [] → α
| []       hxs := by cc
| (x :: _) _   := x
```

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

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

We can then invoke the function as follows:

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

This prints `3`.

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

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

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

The length of a list is defined by recursion:

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

We can say something interesting about the length of 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) :=
begin
  induction xs generalizing ys,
  case list.nil {
    refl },
  case list.cons : x xs ih {
    cases ys,
    case list.nil {
      refl },
    case list.cons : y ys {
      simp [zip, length, ih, min_add_add] } }
end
```

The proof above teaches us yet another trick. The keyword `generalizing` is used here to strengthen the induction hypothesis. Without `generalizing`, the induction hypothesis would be

```
length (zip xs_tl ys) = min (length xs_tl) (length ys)
```

where `ys` is the same list as in the lemma statement. With `generalizing`, the variable is bound by a ∀-quantifier and can be instantiated:

```
∀ys, length (zip xs_tl ys) = min (length xs_tl) (length ys)
```

This is necessary here because we want to instantiate the quantifier with `ys`'s tail and not with `ys` itself.

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 :=
begin
  cases classical.em (m ≤ n),
  case or.inl : h {
    simp [min, h] },
```

```
    case or.inr : h {
      simp [min, h] }
end
```

Recall that `min` is defined by `min a b = (if a ≤ b then a else b)`. To reason about it, we typically need to perform a case distinction on the condition `a ≤ b`. This is achieved using `cases classical.em (a ≤ b)`. This creates two subgoals: one in which `a ≤ b` appears as a hypothesis and one where `¬ a ≤ b` appears.

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

```
lemma min_add_add₂ (l m n : ℕ) :
  min (m + l) (n + l) = min m n + l :=
match classical.em (m ≤ n) with
| or.inl h := by simp [min, h]
| or.inr h := by simp [min, h]
end

lemma min_add_add₃ (l m n : ℕ) :
  min (m + l) (n + l) = min m n + l :=
if h : m ≤ n then
  by simp [min, h]
else
  by simp [min, h]
```

We see again that the mechanisms that are available to write functional programs, such as `match` and `if–then–else`, are also available for writing structured proofs (which are, after all, terms). We can now add a couple of rows to the table at the end of Section 3.7:

Tactical proof	Structured proof	Raw proof term
`cases t,`	`match t with _ end`	`match t with _ end`
`cases classical.em Q,`	`if Q then _ else _`	`if Q then _ else _`

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

```
lemma map_zip {α α' β β' : Type} (f : α → α') (g : β → β') :
  ∀xs ys,
    map (λab : α × β, (f (prod.fst ab), g (prod.snd ab)))
      (zip xs ys) =
    zip (map f xs) (map g ys)
| (x :: xs) (y :: ys) := by simp [zip, map, map_zip xs ys]
| []        _         := by refl
| (_ :: _)  []        := by refl
```

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

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

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

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

```
def f₂ {α : Type} : list α → ···
| []        := ...
| (x :: xs) := ... (x :: xs) ...
```

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

4.8 Binary Trees

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

```
inductive btree (α : Type) : Type
| 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 `tree.empty`, allowing us to write `tree.empty` instead of `tree.empty α`, `tree.empty ℕ`, etc.)

A peculiarity of binary trees is that structural induction gives rise to two induction hypotheses: one for the left subtree of an inner node and one for the right subtree. To prove a goal `t : tree α ⊢ P[t]` by structural induction on `t`, we need to show the subgoals

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

The tree counterpart to list reversal is the mirror operation:

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

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

```
lemma mirror_mirror {α : Type} (t : btree α) :
  mirror (mirror t) = t :=
begin
  induction t,
```

```
  case btree.empty {
    refl },
  case btree.node : a l r ih_l ih_r {
    simp [mirror, ih_l, ih_r] }
end
```

A more detailed informal proof would be as follows:

The proof is by structural induction on `t`.

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

CASE `tree.node a l r`: The induction hypotheses are

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

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

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

□

If we wanted to achieve the same level of detail in the Lean proof, we could always use a calculational block (Section 3.4) instead of `simp`:

```
lemma mirror_mirror₂ {α : Type} :
  ∀t : btree α, mirror (mirror t) = t
| btree.empty        := by refl
| (btree.node a l r) :=
  calc  mirror (mirror (btree.node a l r))
      = mirror (btree.node a (mirror r) (mirror l)) :
    by refl
  ... = btree.node a (mirror (mirror l)) (mirror (mirror r)) :
    by refl
  ... = btree.node a l (mirror (mirror r)) :
    by rewrite mirror_mirror₂ l
  ... = btree.node a l r :
    by rewrite mirror_mirror₂ r
```

4.9 Case Distinction and Induction Tactics

The following tactics are useful when performing case distinctions, with or without induction hypotheses.

cases

```
cases term [with name₁ ... nameₙ]
```

The `cases` tactic performs a case distinction on the specified term. This gives rise to as many subgoals as there are constructors in the definition of the term's type. The tactic behaves roughly the same as `induction` except that it does not produce induction hypotheses and it eliminates impossible cases.

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

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

The `cases` tactic can also be used on a hypothesis `h` of the form `l = r`. It matches `r` against `l` and replaces all occurrences of the variables occurring in `r` with the corresponding terms in `l` everywhere in the goal. The remaining hypothesis `l = l` can be removed using `clear h` if desired. If `r` fails to match `l`, no subgoals emerge; the proof is complete.

```
cases classical.em (proposition) [with name₁ name₂]
```

The `cases` tactic can also be used to perform a case distinction on a proposition. Two cases emerge: one in which the proposition is true and one in which it is false. The optional names *name*$_1$ and *name*$_2$ are used for hypotheses in the true and false cases, respectively.

induction ... generalizing

```
induction term generalizing variables [with name₁ ... nameₙ]
```

The `induction ... generalizing` tactic generalizes the specified variables before performing the induction, resulting in stronger, ∀-quantified induction hypotheses. The tactic otherwise behaves like `induction` without the `generalizing` keyword (Section 2.7).

case

```
case name [: name₁ ... nameₙ] {
  ... }
```

The `case` tactic can be used in conjunction with `cases` and `induction` to select a subgoal corresponding to the constructor called *name* and focus on it.

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

4.10 Dependent Inductive Types

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

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

Thus, the term `vec.cons 3 (vec.cons 1 vec.nil)` has type `vec ℕ 2`. By encoding the vector length in the type, we can provide more precise information about the result of functions. A function such as `vec.reverse`, which reverses a vector, would map a value `vec α n` to another value of the same type, with the same `n`. And `vec.zip` could require its two arguments to have the same length. Fixed-length vectors and matrices are also useful in mathematics.

Unfortunately, this more precise information comes at a cost. Dependent types often make definitions and proofs more difficult, so in this course we will tend to avoid them as much as possible. They are briefly covered here for completeness. To put it in a completely unambiguous way: This is not exam material.

The definitions below introduce conversions between lists and vectors:

```
def list_of_vec {α : Type} : ∀{n : ℕ}, vec α n → list α
| _ vec.nil        := []
| _ (vec.cons a v) := a :: list_of_vec v

def vec_of_list {α : Type} :
  ∀xs : list α, vec α (list.length xs)
| []        := vec.nil
| (x :: xs) := vec.cons x (vec_of_list xs)
```

The `list_of_vec` conversion takes a type `α`, a length `n`, and a vector of length `n` over `α` as arguments and returns a list over `α`. Although we do not care about the length `n`, it still needs to be an argument because it appears in the type of the third argument. The first two arguments are implicit, which is reasonable since they can be inferred from the type of the third argument. The syntax `∀{...}` is new but should be self-explanatory.

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

By the Curry–Howard correspondence, proofs are carried out in much the same way. Let us verify that when converting a vector to a list, we do not accidentally change its length:

```
lemma length_list_of_vec {α : Type} :
  ∀{n : ℕ} (v : vec α n), list.length (list_of_vec v) = n
| _ vec.nil        := by refl
| _ (vec.cons a v) :=
  by simp [list_of_vec, length_list_of_vec v]
```

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

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

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

```
                                          ⊢ P₀[vec.nil]
m : ℕ, a : α, u : vec α m, ih : Pₘ[u] ⊢ Pₘ₊₁[vec.cons a u]
```

Proofs by case distinction using the `cases` tactic work in much the same way, but without the induction hypothesis. Often, the length `n` will be not a variable but a complex term. Then the replacement of `n` in `P[ ]` might not be intuitively meaningful. With `cases`, the corresponding subgoal is simply eliminated. Thus, a case distinction on a value of type `vec α 0` will yield only one subgoal, of the form `⊢ P[vec.nil]`, since `0` could never be equal to a term of the form `m + 1`.

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

```
match v with
| vec.nil      := ...
| vec.cons a u := ...
end
```

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

```
match n, v with
| 0,     vec.nil      := ...
| m + 1, vec.cons a u := ...
end
```

Showing the implicit arguments, we have

```
match n, v with
| 0,     @vec.nil α        := ...
| m + 1, @vec.cons α a m u := ...
end
```

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

```
match n, v with
| _, vec.nil      := ...
| _, vec.cons a u := ...
end
```

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

Incidentally, Lean's core libraries define a type of fixed-length vectors, called `vector α n`, as a subtype of `list α`:

```
def vector (α : Type) (n : ℕ) :=
{xs : list α // list.length xs = n}
```

In other words, the type `vector α n` consists of the values `xs` of type `list α` such that `xs` is of length `n`. This makes it possible to reuse Lean's extensive list library. Subtyping will be explained in Chapter 11.

4.11 Summary of New Constructs

Declaration

`structure`	introduces a structure type and its selectors

Attributes

`@[class]`	declares a structure type as a type class
`@[instance]`	registers a structure value as a type class instance

Term Language

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

Tactics

`case`	focuses on a subgoal and names its variables and hypotheses
`cases`	performs a case distinction
`induction ... generalizing ...`	generalizes the induction hypotheses over the specified terms

Chapter 5

Inductive Predicates

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

> Lean = typed functional programming + logic programming + more logic

5.1 Introductory Examples

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

5.1.1 Even Numbers

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

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

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

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

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

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

A proof is then a derivation tree:

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

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

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

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

This should look familiar. We have used the exact same syntax, except with `Type` instead of `Prop`, to define inductive types. Inductive types and inductive predicates constitute the same mechanism in Lean.

The above command defines a unary predicate called `even` as well as two introduction rules called `even.zero` and `even.add_two` that can be used to prove goals of the form `⊢ even ...`. Recall that an introduction rule for a symbol (e.g., `even`) is a lemma whose conclusion contains that symbol. By the Curry–Howard correspondence, `even n` can be viewed as a dependent inductive type like `vec α n` (Section 4.10), and `even.zero` and `even.add_two` as constructors like `vec.nil` and `vec.cons`.

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

```
lemma even_4 :
  even 4 :=
have even_0 : even 0 :=
  even.zero,
have even_2 : even 2 :=
  even.add_two _ even_0,
show even 4, from
  even.add_two _ even_2
```

The proof term `even.add_two _ even_0` has type `even (0 + 2)`, which is equal to `even 2` up to computation, and similarly for `even.add_two _ even_2`. The underscores stand for `0` and `2`.

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

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

```
constant even : ℕ → Prop
```

[1] We could cheat and add an axiom stating `even 1`, but this would be inconsistent. Alternatively, we could add an assumption `even 1` to our lemma and use it to prove `even 1`, but this would be tautological and hence pointless.

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

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

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

```
def even₂ : ℕ → bool
| 0       := tt
| 1       := ff
| (k + 2) := even₂ k
```

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

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

```
def even₃ (k : ℕ) : bool :=
k % 2 = 0
```

In fact, mathematicians would probably find this version the most satisfactory. But the inductive version is a convenient "Hello, World!" example that is typical of many realistic inductive definitions. It is a toy, but it is a useful toy.

5.1.2 Tennis Games

Transition systems consists of transition rules, which connect a "before" and an "after" state. As a simple specimen of a transition system, we consider the possible transitions in a game of tennis, starting from 0–0 ("Love all"). Tennis games are also a toy, but in Chapter 8, we will define the semantics of an imperative programming language as a transition system in a similar style.

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

> A standard game is scored as follows with the server's score being called first:
>
> | | | |
> |---|---|---|
> | No point | – | "Love" |
> | First point | – | "15" |
> | Second point | – | "30" |
> | Third point | – | "40" |
> | Fourth point | – | "Game" |

> except that if each player/team has won three points, the score is "Deuce." After "Deuce," the score is "Advantage" for the player/team who wins the next point. If that same player/team also wins the next point, that player/team wins the "Game"; if the opposing player/team wins the next point, the score is again "Deuce." A player/team needs to win two consecutive points immediately after "Deuce" to win the "Game."

We first define an inductive type to represent the possible scores:

```
inductive score : Type
| vs       : ℕ → ℕ → score
| adv_srv  : score
| adv_rcv  : score
| game_srv : score
| game_rcv : score
```

A score such as 30–15 is represented as `score.vs 30 15`, which we abbreviate to `30–15`. We ignore some of the most frivolous aspects of the scoring rules, writing `0` for "Love" and `40–40` for "Deuce." If we really cared, we could introduce definitions (e.g., `def love : ℕ := 0`).

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

```
inductive step : score → score → Prop
| srv_0_15    : ∀n, step (0–n) (15–n)
| srv_15_30   : ∀n, step (15–n) (30–n)
| srv_30_40   : ∀n, step (30–n) (40–n)
| srv_40_game : ∀n, n < 40 → step (40–n) score.game_srv
| srv_40_adv  : step (40–40) score.adv_srv
| rcv_0_15    : ∀n, step (n–0) (n–15)
| rcv_15_30   : ∀n, step (n–15) (n–30)
| rcv_30_40   : ∀n, step (n–30) (n–40)
| rcv_40_game : ∀n, n < 40 → step (n–40) score.game_rcv
| rcv_40_adv  : step (40–40) score.adv_rcv
```

Let `s ⇒ t` abbreviate `step s t`. A game is a chain $s_0 \Rightarrow s_1 \Rightarrow s_2 \Rightarrow \cdots \Rightarrow s_n$ where $s_0 = 0–0$ and no transition is possible from s_n. The predicate will allow nonsensical transitions such as `15–99 ⇒ 30–99`, but since the state `15–99` cannot be reached from `0–0`, such transitions are harmless.

Equipped with a formal definition, we can ask, and formally answer, questions such as: Is this transition system confluent? Does it always terminate? How many different final states are possible? And is it actually true that the score 15–99 cannot be reached from "Love all"?

5.1.3 Reflexive Transitive Closure

Are there any convincing non-toy applications of inductive predicates? The answer is yes. Consider the reflexive transitive closure r^* of a binary relation r. The star (*) operator is often defined as a formal system:

$$\frac{(a, b) \in r}{(a, b) \in r^*}\ \textsc{Base} \qquad \frac{}{(a, a) \in r^*}\ \textsc{Refl} \qquad \frac{(a, b) \in r^* \qquad (b, c) \in r^*}{(a, c) \in r^*}\ \textsc{Trans}$$

These rules define r^* as the smallest relation that contains r (by Base) and that is reflexive (by Refl) and transitive (by Trans). If we wanted to define the transitive closure r^+ instead, we would simply omit the Refl rule. If we wanted the reflexive symmetric closure, we would replace the Trans rule by a Symm rule. With a formal system, we simply declare the properties we want to be true, without giving a thought to termination or executability.

It is straightforward to translate the above derivation rules into introduction rules of an inductive predicate:

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

We represent relations as binary predicates rather than sets of pairs, writing `r a b` for `(a, b) ∈ r`. The reflexive transitive closure of `r` is called `star r`. Inside the definition, the argument `r` is omitted, since it is declared as a parameter, on the left of a colon (`:`). Notice that `a`, `b`, and `c` are declared as parameters of the introduction rules, on the left of the colons. We could also have written

```
| base : ∀a b : α, r a b → star a b
```

or at the other extreme

```
| base (a b : α) (hab : r a b) : star a b
```

and similarly for `star.refl` and `star.trans`. This would make no difference.

The general format of inductive predicates is as follows:

```
inductive predicate-name (params₁ : type₁) ... (paramsₖ : typeₖ) :
  typeₖ₊₁ → ··· → type₁ → Prop
| rule-name₁ (params₁₁ : type₁₁) ... (params₁ₘ : type₁ₘ) : proposition₁
    ⋮
| rule-nameₙ (paramsₙ₁ : typeₙ₁) ... (paramsₙₘ : typeₙₘ) : propositionₙ
```

where the conclusion of each $\mathit{proposition}_j$ must be an application of the defined predicate *predicate-name* to some arguments. These arguments may be arbitrary terms and not necessarily constructor patterns. We can also use curly braces `{ }` instead of parentheses `( )` if we wanted to make the arguments corresponding to the parameters implicit.

The above definition of `star` is truly elegant. If you still doubt this, try implementing it as a recursive function:

```
def star₂ {α : Type} (r : α → α → Prop) : α → α → Prop :=
```

5.1.4 A Nonexample

Not all inductive definitions admit a least solution. The simplest nonexample is

```
-- fails
inductive illegal : Prop
| intro : ¬ illegal → illegal
```

If Lean accepted this definition, we could use it to prove `illegal ↔ ¬ illegal`, from which we could easily derive `false` (e.g., using `cc`). Fortunately, Lean rejects the definition:

```
arg #1 of 'illegal.intro' has a non positive occurrence of
the datatypes being declared
```

The nonpositive occurrence it complains about is the occurrence of `illegal` under a negation. Mathematicians would reject the definition on the ground that the monotonicity condition of Knaster–Tarski theorem is not satisfied.

5.2 Logical Symbols

Although `even` is the first openly inductive predicate in this guide, the earlier chapters already presented other inductive predicates clandestinely. The very first of these is equality (`=`), introduced in Chapter 1, followed by the logical symbols ∧, ∨, ↔, ∃, `true`, and `false`. Their definitions are worth studying closely:

```
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 iff (a b : Prop) : Prop
| intro : (a → b) → (b → a) → iff

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

inductive true : Prop
| intro : true

inductive false : Prop

inductive eq {α : Type} : α → α → Prop
| refl : ∀a : α, eq a a
```

(Strictly speaking, in Lean, some of the above definitions are actually structures and not inductive predicates, but the difference is unimportant: Structures are essentially single-constructor inductive predicates with some syntactic sugar.)

The traditional notations `∃x : α, p` and `x = y` are syntactic sugar for `Exists (λ x : α, p)` and `eq x y`. Notice how λ plays the role of an all-purpose binder. Notice also that there is no constructor for `false`. There are no proofs of `false`, just like there is no proof of `even 1`. With inductive predicates, we only state the rules we want to be true.

The symbol ∀, including its special case →, is built directly into the logic and is not defined as inductive predicates. Nor does it have explicit introduction or elimination rules. The introduction principle is λ-expression `λx, _`, and the elimination principle is function application `_ u`.

As for any inductive predicates, only the introduction rules are specified. The elimination rules presented in Sections 2.3 and 2.4 must be derived manually.

5.3 Rule Induction

In the same way that we can perform an induction on a term of inductive type, we can perform an induction on a proof of an inductive predicate. For example, given the goal `h : even n ⊢ P`, we can invoke the tactic `induction h` and obtain two subgoals, corresponding to `even.zero` and `even.add_two`. This is called *induction on the structure of the derivation of* `h` or simply *rule induction*, because the induction is on the predicate's introduction rules (i.e., the constructors of the proof term). There are two ways to look at rule induction: the least-predicate-such-that view and the Curry–Howard view.

To understand the least-predicate-such-that view, recall that an inductive definition introduces a symbol as the least (i.e., the most false) predicate satisfying the specified introduction rules. Accordingly, `even` is the least predicate `q` such that the properties `q 0` and `∀k, q k → q (k + 2)` hold. Therefore, if we can show that `p 0` and `∀k, p k → p (k + 2)` hold for some predicate `p`, then `p` is either `even` itself or greater than (i.e., more true than) `even`. This means that `even n` implies `p n`, which is exactly what we need to prove the goal `h : even n ⊢ p n`.

The least-predicate-such-that view gives a nice intuitive account of rule induction that can be used in informal arguments, such as the following proof that `even n` implies `n % 2 = 0` for all `n`:

> The proof is by rule induction on the hypothesis `even n`.
>
> Case `even.zero`: We must show `0 % 2 = 0`. This follows by computation.
>
> Case `even.add_two k`: The induction hypothesis is `k % 2 = 0`. We must show `(k + 2) % 2 = 0`. This follows by basic arithmetic reasoning. □

The Lean proof has the same structure:

```
lemma mod_two_eq_zero_of_even (n : ℕ) (h : even n) :
  n % 2 = 0 :=
begin
  induction h,
  case even.zero {
    refl },
  case even.add_two : k hk ih {
    simp [ih] }
end
```

The Curry–Howard correspondence gives us another fruitful way to look at rule induction. Essentially, rule induction on `h` in on a goal such as `h : even n → P[h]` is perfectly analogous to structural induction on a value of a dependent inductive type such as `vec α n` (Section 4.10). Using the notation $P_u[\,]$ for the variant of `P[ ]` where all occurrences of `n` are replaced by term `u`, the subgoals are

$$\vdash P_0[\texttt{even.zero} : \texttt{even } 0]$$

$$\texttt{k} : \mathbb{N}, \texttt{hek} : \texttt{even k}, \texttt{ih} : P_k[\texttt{hk}] \vdash P_{k+2}[\texttt{even.add\_two k hk} : \texttt{even (k + 2)}]$$

These are precisely the subgoals produced by `induction h with k hk ih`.

Regardless of the inductive predicate `q`, the procedure to compute the subgoals is always the same:

1. Replace `h` in `P[h]` with each possible introduction rule applied to fresh variables (e.g., `even.add_two k hk`), massaging `P[ ]` to make it type-correct. This yields as many subgoals as there are introduction rules.
2. Add these new variables (e.g., `k`, `hek`) to the local context.
3. Add induction hypotheses for all new hypotheses asserting `q` ....

Notice the presence of `hk : even k` among the hypotheses. It was absent in the least-predicate-such-that view and is not strictly necessary because `P` can always be strengthened to be of the form `even n` $\wedge \cdots$.

In nearly all practical cases, `h` will not occur in `P[h]`. We can then simply write

$$\vdash P_0 \qquad\qquad k : \mathbb{N},\ \mathtt{hek} : \mathtt{even}\ k,\ \mathtt{ih} : P_k \vdash P_{k+2}$$

In rare cases, `h` will occur in `P[h]`. Proofs may appear as subterms in arbitrary terms, as we saw when we tried to extract the head of a list in Section 4.7.

The reflexive transitive closure `star r` follows the same scheme. Given a goal `h : star r x y` $\vdash$ `P`, rule induction on `h` produces the following subgoals, where $P_{t,u}$ denotes the variant of `P` where `x` and `y` are replaced by `t` and `u`, respectively:

$$\mathtt{a\ b} : \alpha,\ \mathtt{hab} : r\ a\ b \vdash P_{a,b}$$

$$\mathtt{a} : \alpha \vdash P_{a,a}$$

$$\mathtt{a\ b\ c} : \alpha,\ \mathtt{hab} : \mathtt{star}\ r\ a\ b,\ \mathtt{hbc} : \mathtt{star}\ r\ b\ c,\ \mathtt{ihab} : P_{a,b},\ \mathtt{ihbc} : P_{b,c} \vdash P_{a,c}$$

This looks rather formidable. Fortunately, this is where the "assistant" aspect of "proof assistant" comes into play. One of the key properties of `star` is that it is idempotent—applying `star` to `star r` has no effect. This can be proved as follows in Lean, using rule induction for the $\to$ direction of the equivalence:

```
lemma star_star_iff_star {α : Type} (r : α → α → Prop)
    (a b : α) :
  star (star r) a b ↔ star r a b :=
begin
  apply iff.intro,
  { intro h,
    induction h,
    case star.base : a b hab {
      exact hab },
    case star.refl : a {
      apply star.refl },
    case star.trans : a b c hab hbc ihab ihbc {
      apply star.trans a b,
      { exact ihab },
      { exact ihbc } } },
  { intro h,
    apply star.base,
    exact h }
end
```

We use the `case` tactic both to document which cases we are focusing on and to give intuitive names to the emerging variables. It is easy to get lost in goals containing long, automatically generated names. The cleanup tactics introduced in Section 2.8 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:

```
@[simp] lemma star_star_eq_star {α : Type}
    (r : α → α → Prop) :
  star (star r) = star r :=
begin
  apply funext,
  intro a,
  apply funext,
  intro b,
  apply propext,
  apply star_star_iff_star
end
```

The proof requires two lemmas that are available because Lean's logic is classical:

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

Functional extensionality (`funext`) states that two functions that yield equal results for all inputs must be equal. *Propositional extensionality* (`propext`) states that equivalence of propositions coincides with equality. These properties may seem obvious, but there are proof assistants built on weaker, intuitionistic logics in which the property do not generally hold.

We register the lemma `star_star_eq_star` as a simp rule, because viewed as a left-to-right rewrite rule, it genuinely replaces a complex term by a simpler term. It is hard to imagine a situation where we would not want `simp` to rewrite `star (star ...)` to `star ...`.

For rule induction, we normally use the `induction` tactic. For subtle logical reasons that will become clearer in Chapter 11, rule induction by pattern matching is generally not possible.

5.4 Rule Induction Pitfalls

Inductive predicates often have arguments that evolve over the course of the induction. Some care is necessary when invoking the `induction` tactic. Such details are often glossed over in informal proofs, but proof assistants require us to be precise.

Recall the inductive definition of even numbers:

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

If the goal has the form `h : even n ⊢ P`, invoking `induction h with k hek ih` produce the following subgoals:

$$\vdash P_0 \qquad\qquad k : \mathbb{N},\ \mathtt{hek} : \mathtt{even}\ k,\ \mathtt{ih} : P_k \vdash P_{k+2}$$

This works as desired.

The problem is that when the argument to `even` is not a variable, `induction` destroys its structure. With a variable, there is no structure destroy. Thus, given the goal `h : even (2 * n + 1) ⊢ false`, invoking `induction h with k hk ih` produces

$$\vdash \mathtt{false} \qquad\qquad k : \mathbb{N},\ \mathtt{hek} : \mathtt{even}\ k,\ \mathtt{ih} : \mathtt{false} \vdash \mathtt{false}$$

The first subgoal is obviously not provable. To avoid this, we need to rephrase the lemma statement, replacing `2 * n + 1` by a fresh variable `x` and adding an equation `2 * n + 1 = x` as a hypothesis. The new goal

$$n\ x : \mathbb{N},\ \mathtt{hx} : 2 * n + 1 = x,\ \mathtt{hex} : \mathtt{even}\ x \vdash \mathtt{false}$$

is equivalent to the original goal, but it yields very different subgoals:

$$\mathtt{hx} : 2 * n + 1 = 0 \vdash \mathtt{false}$$
$$n\ k : \mathbb{N},\ \mathtt{hx} : 2 * n + 1 = k + 2,\ \mathtt{hek} : \mathtt{even}\ k,\ \mathtt{ih} : 2 * n + 1 = k \to \mathtt{false} \vdash \mathtt{false}$$

We can now eliminate the first subgoal by noting that the hypothesis is false. Unfortunately, the second subgoal is just as difficult to prove as the original goal. The induction hypothesis is useless because there is no hope to discharge the assumption `2 * n + 1 = k` using `2 * n + 1 = k + 2`. If only we could instantiate `n` in the induction hypothesis with `n - 1`, we would have `2 * (n - 1) + 1 = k + 2`, but `n` cannot be instantiated.

The solution is to explicitly quantify over `n` so that we can instantiate it in the induction hypothesis. We state our initial goal as

$$x : \mathbb{N},\ \mathtt{hex} : \mathtt{even}\ x \vdash \forall n,\ x = 2 * n + 1 \to \mathtt{false}$$

This time, `induction x` produces the subgoals

$$\mathtt{hx} : 2 * n + 1 = 0 \vdash \mathtt{false}$$
$$n\ k : \mathbb{N},\ \mathtt{hx} : 2 * n + 1 = k + 2,\ \mathtt{hek} : \mathtt{even}\ k,\ \mathtt{ih} : (\forall n,\ 2 * n + 1 = k \to \mathtt{false}) \vdash \mathtt{false}$$

Now the variable `n` in `hx` is disconnected from the variable `n` in the induction hypothesis, and we can instantiate the induction hypothesis's `n` with `n - 1`.

Putting all of this together, we obtain the following proof:

```
lemma not_even_2_mul_add_1 (n : ℕ) :
  ¬ even (2 * n + 1) :=
begin
  generalize hx : 2 * n + 1 = x,
  intro h,
  induction h generalizing n,
  case even.zero {
    cases hx },
  case even.add_two : k hk ih {
```

```
      apply ih (n - 1),
      cases n,
      case nat.zero {
        linarith },
      case nat.succ : m {
        simp [nat.succ_eq_add_one] at *,
        linarith } }
end
```

The `generalize hx : t = x` tactic transforms a goal of the form `Q, R[t] ⊢ S[t]` into `Q, hx : t = x, R[x] ⊢ S[x]`, where all occurrences of the term `t` are replaced by the fresh variable `x`. The `generalizing` option to `induction` (Section 4.2) can be used to quantify over a variable so that we can instantiate it in the induction hypothesis. In conjunction, these two mechanisms make it possible to perform induction without needing to modify the statement of the lemma to prove.

5.5 Miscellaneous Tactics

generalize

```
generalize name : term = variable
```

The `generalize` tactic replaces all occurrences of *`term`* in the goal with a fresh *`variable`* and introduces a hypothesis *`term`* `=` *`variable`* called *`name`*. This can be used before performing an induction to replace concrete terms with variables. This tactic should not be confused with the `generalizing` option of the `induction` tactic (Section 2.7), which can be used to strengthen the induction hypotheses with ∀-quantifiers.

linarith

The `linarith` tactic can be used to prove goals involving linear arithmetic equalities (`=`), inequalities (`<`, ≤, `>`, and ≥), and disequalities (≠). *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 (which can be rewritten to `x + x < y`) whereas `x * y < y` is nonlinear.

5.6 Elimination

Given an inductive predicate `q`, its introduction rules typically are of the form ∀..., ⋯ → `q` ... and can be used to prove goals of the form ⊢ `q` .... Elimination works the other way around: It extracts information from a lemma or hypothesis `h : q` .... Elimination takes many forms: the `cases` and `induction` tactics, pattern matching, and custom elimination rules (e.g., `and.elim_left`).

Invoked on `h : q` ..., the `cases h` tactic performs roughly the same rule induction as `induction h` but without producing any induction hypotheses. We encountered two idioms in Chapter 4 that we can finally understand.

The first idiom is when `h` is of the form `l = r`—i.e., `eq l r` (Section 5.2). Suppose the goal is `h : l = r ⊢ P[h]`. The procedure presented in Section 5.3 produces the

subgoal

$$\texttt{a} : \alpha \vdash \texttt{P}_{\texttt{a},\texttt{a}}\texttt{[eq.refl a : a = a]}$$

in which $\texttt{P}_{\texttt{t},\texttt{u}}\texttt{[ ]}$ stands for the variant of `P[ ]` where `l` and `r` are replaced by `t` and `u`, respectively. (Strictly speaking, the useless hypothesis `h : a = a` would also appear in the subgoal.) In practice, `P[h]` would likely not depend on `h`. Moreover, `cases` reuses the name `l` instead of inventing the name `a`. Thus, we would get

$$\texttt{l} : \alpha \vdash \texttt{P}_{\texttt{l},\texttt{l}}$$

In other words, all occurrences of `r` in the original goal have been replaced by `l`. This corresponds to the behavior we observed in Section 4.7.

The second idiom is the tactic `cases classical.em Q`, where `Q` is a proposition. The `classical.em Q` part is a proof term for $\texttt{Q} \vee \neg\, \texttt{Q}$—i.e., `or Q (¬ Q)` (Section 5.2). Then `cases` is simply applied to eliminate the $\vee$ connective. Suppose the goal is `⊢ P[classical.em Q]`. By the definition of the `or` predicate, the new subgoals are

$$\texttt{hQ} : \texttt{Q} \vdash \texttt{P[or.intro\_left hQ : Q} \vee \neg\, \texttt{Q]}$$
$$\texttt{hnQ} : \neg\, \texttt{Q} \vdash \texttt{P[or.intro\_right hnQ : Q} \vee \neg\, \texttt{Q]}$$

There is no need to massage `P` because `or.intro_left hQ` and `or.intro_right hnQ` have the same type as `classical.em Q`—namely, $\texttt{Q} \vee \neg\, \texttt{Q}$. In practice, `P` would likely not depend on `classical.em Q`. We would then have

$$\texttt{hQ} : \texttt{Q} \vdash \texttt{P} \qquad\qquad \texttt{hnQ} : \neg\, \texttt{Q} \vdash \texttt{P}$$

Again, this is the behavior we observed in Section 4.7.

In structured proofs, we can use `match` expressions (Section 4.4) to achieve the same effect as `cases`. This works well for logical symbols. However, for predicates such as `even` and `star`, with arguments that evolve through the induction, we end up with dependently typed pattern matching, which is subtle (Section 4.10). It is generally easier to let `cases` figure out what the subgoals should look like than to pattern-match. We will review an example of both styles below.

Often it is convenient to expand a hypothesis of the form `q (c ...)`, where `c` is a constructor or some other constant. We can state and prove an *inversion rule* to support such eliminative reasoning. A typical inversion rule has the form

$$\forall \texttt{x}_1 \ldots \texttt{x}_n,\ \texttt{q}\ (\texttt{c}\ \texttt{x}_1 \ldots \texttt{x}_n) \to (\exists \ldots, \cdots \wedge \cdots) \vee \cdots \vee (\exists \ldots, \cdots \wedge \cdots)$$

It can be useful to combine introduction and elimination into a single lemma, which can be used for rewriting both the hypotheses and the targets of goals. The format is the same except for the connective $\leftrightarrow$ in the middle:

$$\forall \texttt{x}_1 \ldots \texttt{x}_n,\ \texttt{q}\ (\texttt{c}\ \texttt{x}_1 \ldots \texttt{x}_n) \leftrightarrow (\exists \ldots, \cdots \wedge \cdots) \vee \cdots \vee (\exists \ldots, \cdots \wedge \cdots)$$

An inversion rule for `even` would be

```
lemma even_iff (n : ℕ) :
  even n ↔ n = 0 ∨ (∃m : ℕ, n = m + 2 ∧ even m) :=
begin
  apply iff.intro,
  { intro hn,
```

```
    cases hn,
    case even.zero {
      simp },
    case even.add_two : k hk {
      apply or.intro_right,
      apply exists.intro k,
      simp [hk] } },
  { intro hor,
    cases hor,
    case or.inl : heq {
      simp [heq, even.zero] },
    case or.inr : hex {
      cases hex with k hand,
      cases hand with heq hk,
      simp [heq, even.add_two _ hk] } }
end
```

As usual, the tactical proof is not particularly readable, but we see that introduction rules and the eliminative `cases` tactic play a major role, for both the logical symbols and the `even` predicate. The `simp` tactic puts the final touches.

For those who have a preference for structured proofs, here is a version of the proof with dependently typed pattern matching on `hn : even n`:

```
lemma even_iff₂ (n : ℕ) :
  even n ↔ n = 0 ∨ (∃m : ℕ, n = m + 2 ∧ even m) :=
iff.intro
  (assume hn : even n,
   match n, hn with
   | _, even.zero         :=
     show 0 = 0 ∨ _, from
       by simp
   | _, even.add_two k hk :=
     show _ ∨ (∃m, k + 2 = m + 2 ∧ even m), from
       or.intro_right _ (exists.intro k (by simp [*]))
   end)
  (assume hor : n = 0 ∨ (∃m, n = m + 2 ∧ even m),
   match hor with
   | or.intro_left _ heq :=
     show even n, from
       by simp [heq, even.zero]
   | or.intro_right _ hex :=
     match hex with
     | Exists.intro m hand :=
       match hand with
       | and.intro heq hm :=
         show even n, from
           by simp [heq, even.add_two _ hm]
       end
     end
   end)
```

5.7 Further Examples

Equipped with a better understanding of inductive predicates, we are now ready to review in turn four realistic applications.

5.7.1 Sorted Lists

Ou first 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} {zs : list ℕ} (hle : x ≤ y)
    (hsorted : sorted (y :: zs)) :
  sorted (x :: y :: zs)
```

This definition captures the following mathematical intuition:

> The set of sorted lists is defined as the smallest set `S` closed under the following rules: (1) The list `[]` is sorted; (2) given a number `x`, the list `[x]` is sorted; (3) given two numbers `x`, `y` and a list `zs`, if `x < y` and `y :: zs` is sorted, then `x :: y :: zs` is sorted.

It is always a good idea to test our definitions by trying it on small examples. Is the list `[3, 5]` sorted? It would appear so:

```
lemma sorted_3_5 :
  sorted [3, 5] :=
begin
  apply sorted.two_or_more,
  { exact dec_trivial },
  { exact sorted.single }
end
```

The example needs two of the introduction rules for `sorted` as well as the special lemma `dec_trivial`, which can be used on trivial decidable goals such as $3 \leq 5$. This approach tends to work well for expressions consisting only of closed terms. A more compact proof follows, using proof terms for everything:

```
lemma sorted_3_5₂ :
  sorted [3, 5] :=
sorted.two_or_more dec_trivial sorted.single
```

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

```
lemma sorted_7_9_9_11 :
  sorted [7, 9, 9, 11] :=
sorted.two_or_more dec_trivial
  (sorted.two_or_more dec_trivial
     (sorted.two_or_more dec_trivial
        sorted.single))
```

Conversely, we can show that some lists are not sorted. For this, we need to use elimination:

```
lemma not_sorted_17_13 :
  ¬ sorted [17, 13] :=
assume h : sorted [17, 13],
have 17 ≤ 13 :=
  match h with
  | sorted.two_or_more hle _ := hle
  end,
have ¬ 17 ≤ 13 :=
  dec_trivial,
show false, from
  by cc
```

To prove a negation, we assume the unnegated proposition and prove `false`. Lean accepts the proof because `¬ P` is computationally equal to `P → false`. From the assumption that `[17, 13]` is sorted, we extract `17 ≤ 13`. The `match` expression can ignore the `sorted.nil` and `sorted.single` cases, because they cannot match a two-element list. Together with the obvious numeric truth that `¬ 17 ≤ 13`, we obtain a contradiction, which is easily detected by `cc`.

5.7.2 Palindromes

Palindromes are lists that read the same from left to right and from right to left. For example, `[a, b, b, a]` and `[a, h, a]` are palindromes. The following inductive predicate is `true` if and only if the list passed as argument is a palindrome:

```
inductive palindrome {α : Type} : list α → Prop
| nil : palindrome []
| single (x : α) : palindrome [x]
| sandwich (x : α) (xs : list α) (hxs : palindrome xs) :
  palindrome ([x] ++ xs ++ [x])
```

The definition distinguishes three cases: (1) `[]` is a palindrome; (2) for any element `x`, the singleton list `[x]` is a palindrome; (3) for any element `x` and any palindrome `[y₁, ..., yₙ]`, the list `[x, y₁, ..., yₙ, x]` is a palindrome.

Palindromes are another example where inductive predicates come into their own. The following naive recursive definition cannot work because `[x] ++ xs ++ [x]` is not a constructor pattern, and the variable `x` is repeated:

```
-- fails
def palindrome₂ {α : Type} : list α → Prop
| []                := true
| [_]               := true
| ([x] ++ xs ++ [x]) := palindrome₂ xs
| _                 := false
```

A correct recursive definition is possible, but it is beyond the scope of this chapter.

It should not come as a surprise that the reverse of a palindrome is also a palindrome. It is a good exercise:

```
lemma reverse_palindrome {α : Type} (xs : list α)
    (hxs : palindrome xs) :
  palindrome (reverse xs) :=
begin
```

```
  induction hxs,
  case palindrome.nil {
    exact palindrome.nil },
  case palindrome.single : x {
    exact palindrome.single x },
  case palindrome.sandwich : x xs hxs ih {
    simp [reverse, reverse_append],
    exact palindrome.sandwich _ _ ih }
end
```

Informally:

> The proof is by rule induction on the hypothesis `hxs`.
>
> Case `palindrome.nil`: We must show `palindrome (reverse [])`. This follows from `palindrome.nil` using `reverse [] = []`.
>
> Case `palindrome.single x`: We must show `palindrome (reverse [x])`. This follows from `palindrome.single` using `reverse [x] = [x]`.
>
> Case `palindrome.sandwich x xs hxs`: We must show `palindrome (reverse ([x] ++ xs ++ [x]))` under the hypothesis (`hxs`) `palindrome xs`. The induction hypothesis is `palindrome (reverse xs)`. By simplification, it suffices to show `palindrome ([x] ++ reverse xs ++ [x])`. By `palindrome.sandwich`, it suffices to show `palindrome (reverse xs)`, which is exactly the induction hypothesis. □

5.7.3 Full Binary Trees

Our third example is based on the type of binary trees introduced in Section 4.8:

```
inductive btree (α : Type) : Type
| empty {} : btree
| node     : α → btree → btree → btree
```

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 btree.empty
| node (a : α) (l r : btree α)
    (hl : is_full l) (hr : is_full r)
    (hiff : l = btree.empty ↔ r = btree.empty) :
  is_full (btree.node a l r)
```

The first case states that the empty tree is a full tree. The second case states that a nonempty tree is a full tree if has two child trees that are themselves full and that are both empty or both nonempty. The two cases neatly follow the structure of the inductive type, so it is natural to reuse the names `empty` and `node`.

The tree that consists of a node with empty 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 (btree.node a btree.empty btree.empty) :=
begin
```

```
  apply is_full.node,
  { exact is_full.empty },
  { exact is_full.empty },
  { refl }
end
```

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

```
lemma is_full_mirror {α : Type} (t : btree α)
    (ht : is_full t) :
  is_full (mirror t) :=
begin
  induction ht,
  case is_full.empty {
    exact is_full.empty },
  case is_full.node : a l r hl hr hiff ih_l ih_r {
    rewrite mirror,
    apply is_full.node,
    { exact ih_r },
    { exact ih_l },
    { simp [mirror_eq_empty_iff, *] } }
end
```

Since `is_full`'s definition follows `btree`'s definition, it is also reasonable to perform structural induction on the tree `t`:

```
lemma is_full_mirror₂ {α : Type} :
  ∀t : btree α, is_full t → is_full (mirror t)
| btree.empty        :=
  begin
    intro ht,
    exact ht
  end
| (btree.node a l r) :=
  begin
    intro ht,
    cases ht with _ _ _ hl hr hiff,
    rewrite mirror,
    apply is_full.node,
    { exact is_full_mirror₂ _ hr },
    { apply is_full_mirror₂ _ hl },
    { simp [mirror_eq_empty_iff, *] }
  end
```

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

5.7.4 First-Order Terms

Our last example is based on an inductive type of first-order terms—the terms of predicate logic:

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

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 $t_1, \ldots, t_n$ stand for the arguments, which are themselves terms. Thus, sin(max(x, y)) is a first-order term. The parameters α and β are the types of function symbols and variables, respectively.

Not all terms are legal. For example, the term min(cos(a), cos(a, b)) is considered ill-formed, because the function cos is invoked with different number of arguments (1 versus 2). Along with α and β, we also consider the arity, represented by a function `arity : α → 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 (term.var x)
| fn (f : α) (ts : list (term α β))
    (hargs : ∀t ∈ ts, well_formed t)
    (hlen : list.length ts = arity f) :
  well_formed (term.fn f ts)
```

The `fn` case checks that the arguments `ts` are recursively well formed and that the length of `ts` equals the specified arity for the function symbol `f` in question.

Another interesting property of first-order terms is whether they contain variables. This can be checked easily using an inductive predicate:

```
inductive 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 (term.fn f ts)
```

There is no introduction rule corresponding to `term.var` because variables are never variable-free.

5.8 Summary of New Constructs

Lemmas

`dec_trivial`	decidable truth (e.g., a true closed executable expression)
`funext`	functional extensionality
`propext`	propositional extensionality

Tactics

`generalize`	replaces a term by a fresh variable defined by a new hypothesis
`linarith`	applies a procedure for linear arithmetic

Chapter 6

Monads

Pure functional programming can sometimes feel overly restrictive. *Effectful functional programming* provides idioms that alleviates some of these restrictions, giving us the impression of programming with side effects, exceptions, nondeterminism, and other effects.

The underlying abstraction is called *monad*. Monads generalize programs with side effects. They are popular in Haskell to write imperative programs. In Lean, they are used to express imperative programs and to reason about them. They are even useful for programming Lean itself, as we will see in Chapter 7.

These notes are inspired by Chapter 7 of *Programming in Lean* [2]. We also refer to Chapter 14 of *Real World Haskell* [25] for a general introduction to effectful functional programming.

6.1 Introductory 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 too few items, return `option.none`.

A straightforward solution would be as follows:

```
def sum_2_5_7 (ns : list ℕ) : option ℕ :=
match list.nth ns 1 with
| option.none    := option.none
| option.some n2 :=
  match list.nth ns 4 with
  | option.none    := option.none
  | option.some n5 :=
    match list.nth ns 6 with
    | option.none    := option.none
    | option.some n7 := option.some (n2 + n5 + n7)
    end
  end
end
```

(Confusingly, `list.nth` counts elements from 0.) The code is quite inelegant, because of all the pattern matching on `option`. Although the programming task is contrived, we can all recall writing code with nested error handling and ever increasing indentation levels.

We can do better, by concentrating all the ugliness in one function:

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

That function operates on an option—a value of type `option α`. If the option is `option.none`, we leave it as is. This corresponds to an error condition, and errors are remembered. Otherwise, the option is of the form `option.some a`, and we apply the operation `f` on `a`—or *bind* `f`'s argument to `a`. We can now use `bind_opt` to program our sum function:

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

Intuitively, the program performs the following steps:

1. Extract the second item from the list. If it is `option.none`, we are done. Otherwise, bind `n2` to this item and continue with the next step.
2. Perform the same for the fifth and seventh item, mutatis mutandis.
3. Return the sum of `n2`, `n5`, and `n7` in an `option.some` wrapper.

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

Instead of defining `bind_opt` 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, pure (n2 + n5 + n7))))
```

We also use the predefined `pure` function instead of `option.some` to convert a pure α value to an option.

One of the advantages of using the predefined notions is that they provide syntactic sugar, in the form of the `>>=` 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, pure (n2 + n5 + n7)
```

The syntax `ma >>= f` expands to `bind ma f`, where `ma` stands for "maybe an α."

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,
      pure (n2 + n5 + n7)
```

The `do` notation provides an intuitive syntax for effectful programs. The syntax `do a ← ma, t` is equivalent to `ma >>= (λa, t)`. If we are not interested in the result of `ma`'s computation, we can omit the `a ←` binding and write `do ma, t`, which expands to `ma >>= (λ_, t)`.

The `do` notation conveniently allows multiple `←` bindings in a single 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,
  pure (n2 + n5 + n7)
```

Each line with an arrow `←` attempts to read a value. In case of failure, the entire program evaluates to `option.none`.

The above function can be read as an imperative program where each of the `list.nth` calls can throw an exception. But even though the notation has an imperative flavor, the function is a pure functional program.

6.2 Two Operations and Three Laws

The `option` type constructor is an example of a monad, called the *option monad*. In general, a monad is a unary type constructor `m : Type → Type` that depends on some type parameter α equipped with two distinguished operations:

```
pure {α : Type} : α → m α
bind {α β : Type} : m α → (α → m β) → m β
```

As usual, curly braces denote implicit arguments. For options, the `pure` operation is simply `option.some`, whereas `bind` is what we called `bind_opt`.

Recall that a value of type `m α` is an effectful program. The `pure` operation embeds a pure, effectless program of type α in `m α`. The `bind` operation composes two effectful programs, of types `m α` and `m β`. The first program's output, of type α, is passed to the second program. The second program's output is also the output of the composite program.

We can think of a monad as a box containing some data. The box captures some special effect (e.g., exceptions, a mutable state). The `pure` operation puts data into the box, whereas `bind` allows us to access the data in the box and modify it—possibly even changing its type, since the result has type `m β`, not `m α`. There is, however, no general way to extract the data from the box—i.e., to obtain an α from an `m α`. There might not be any α value in it, or there might be several.

To summarize, `pure a` provides a box containing the the value `a`, with no side effects, whereas `bind ma f` (also written `ma >>= f` or `do a ← ma, f a`) executes `ma`,

then executes `f` with the boxed result `a` of `ma`. It is convenient to use names such as `ma` or `mb` for monad values of type `m α` or `m β`, and `a` or `b` for data of type `α` or `β`.

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

Type	Effect
`id α`	no effect
`option α`	simple exceptions
`σ → α × σ`	threading through a state of type `σ`
`set α`	nondeterministic computations returning `α` values
`t → α`	reading elements of type `t` (e.g., a configuration)
`ℕ × α`	adjoining running time (e.g., to model algorithmic complexity)
`string × α`	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 proof assistant

All of the above are unary type constructors `m` with an argument `α`. Some effects can be combined (e.g., `option (t → α)`). Some effects are not executable (e.g., `prob α`); they are nonetheless useful for modeling programs abstractly in the logic. Specific type constructors `m` may provide further operators beyond `pure` and `bind`. For example, they may provide a way to extract the boxed value.

Monads have several benefits. They provide the convenient and highly readable `do` notation. They support generic operations, such as `list.mmap {α β : Type} :(α → m β) → list α → m (list β)`, that work uniformly for all monads `m`. To quote *Programming in Lean* [2]:

> The power of the abstraction is not only that it provides general functions and notation that can be used in all these various instantiations, but also that it provides a helpful way of thinking about what they all have in common.

Besides being a useful computer science concept, monads provide a nice example of axiomatic reasoning, the axioms being the monad laws, introduced below.

The `bind` and `pure` operations are normally required to obey three laws. The `bind` operation combines two programs. If either of these is a pure program, we can inline it and eliminate the `bind`. This gives us the first two laws:

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

and

```
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
```

Notice that we can use curly braces `{ }` around the body of a `do` block to delimit it, instead of enclosing the entire block in parentheses `( )`.

The `do` notation can also be used in conjunction with `let`. Instead of

```
do
  a' ← pure a,
  b ← f a',
  g b
```

we can write

```
do
  let a' := a in
    do
      b ← f a',
      g b
```

or

```
do
  let a' := a,
  b ← f a',
  g b
```

6.3 A Type Class

Monads are a mathematical structure, so we use a type class to specify them in Lean. Recall that a type class is a structure type that is parameterized, typically by a type, but here by a type constructor `m : Type → Type`. Whenever we use a field from the type class on a concrete `m`, the type class inference mechanism automatically retrieves the relevant structure value—the type class instance.

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

```
@[class] structure 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`.

- The structure inherits the fields, and any syntactic sugar, from structures called `has_bind` and `has_pure`. These provide the `bind` and `pure` operations on `m`, with the expected types and the syntactic sugar.
- `Type 1` is necessary instead of `Type` for reasons that will become clear 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 monad laws.

We call our type class "lawful monad" because the monad laws are required to hold. To instantiate this definition, we must supply the type constructor `m`, suitable `bind` and `pure` operators, and proofs of the monad 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 a syntactic type class `monad` and a semantic type class `is_lawful_monad`.

6.4 Identity

Lean's constant `id {α : Type} : α → α` is defined as the identity function `λx, x`. The identity type constructor is obtained by taking `α := Type`. We can register it as a monad:

```
def id.pure {α : Type} : α → id α :=
id

def id.bind {α β : Type} : id α → (α → id β) → id β
| a f := f a

@[instance] def id.lawful_monad : lawful_monad (@id Type) :=
{ pure       := @id.pure,
  bind       := @id.bind,
  pure_bind  :=
    begin
      intros α β a f,
      refl
    end,
  bind_pure  :=
    begin
      intros α m,
      refl
    end,
  bind_assoc :=
    begin
      intros α β γ f g m,
      refl
    end }
```

The *identity monad* is the simplest monad possible. It provides a simple box, with a single value in it, without any effect. It plays a similar role as 0 in additive arithmetic. We can think of the other monads as variations of it; for example, the

option monad is an identity monad enriched with a special `option.none` value representing an error state.

6.5 Exceptions

As we saw above, the option monad provides a simple kind of exceptions. The following code extract shows how to register `option : Type → Type` as a lawful monad:

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

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

@[instance] def option.lawful_monad : 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 },
      { refl }
    end,
  bind_assoc :=
    begin
      intros α β γ f g m,
      cases m,
      { refl },
      { refl }
    end }
```

The registration process requires us to provide five components: the `pure` and `bind` operations and the proofs of the three monad laws. 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.

Beyond the standard operations, it can be useful to throw and catch exceptions. This can be implemented as follows:

```
def option.throw {α : Type} : option α :=
option.none
```

```
def option.catch {α : Type} :
  option α → option α → option α
| option.none     ma' := ma'
| (option.some a) _   := option.some a
```

The `option.throw` operation raises an exception, leaving the program in an error state (`option.none`). The `option.catch` operation can be used to recover from an earlier exception. If the program is currently in an error state, `option.catch` invokes some exception-handling code (its second argument). This code might in turn raise a new exception. If `option.catch` is applied to a normal state (of the form `option.some a`), nothing happens.

As a convenient alternative to `option.catch ma ma'`, Lean supports the syntax `ma.catch ma'`. Here is a schematic example that demonstrates throwing and catching with this syntax:

```
do {
  ...,
  if ... then
    option.throw
  else
    ... }
.catch do {
  ... }
```

The corresponding Java code would look as follows:

```
try {
    ...
    if (...) {
        throw new UnknownException();
    } else {
        ...
    }
} catch (UnknownException e) {
    ...
}
```

A more idiomatic approach is to register `option.catch` as the option type's "or else" operator `<|>`:

```
@[instance] def option.has_orelse : has_orelse option :=
{ orelse := @option.catch }
```

This enables us to write

```
do {
  ...,
  if ... then
    option.throw
  else
    ... }
<|>
```

```
do {
  ... }
```

Monads supporting the <|> operator are called *alternative monads*.

Options support only one kind of error state. A more general abstraction, called *error monad*, allows us to distinguish between different errors, as with the exceptions of Java and other programming languages.

6.6 Mutable State

The *state monad* provides an abstraction corresponding to a mutable state. For some programming languages, the compiler can detect the use of the state monad and translate programs using them to more efficient imperative programs.

Admittedly, "abstraction corresponding to a mutable state" may sound somewhat abstract, so let us consider a semiconcrete example. If you have some experience with functional programming, you probably at some point or other have written code that looks very much like this fragment:

```
def welcome_new_user (user_name : string) (ctxt : context) :
  (ℕ × string) × context :=
let
  (user, ctxt') := create_user user_name ctxt,
  (password, ctxt'') := generate_temporary_password user ctxt',
  (ok, ctxt''') := send_unencrypted_email user password ctxt''
in
  ((user, password), ctxt''')
```

(It is very important to send the password in an unencrypted email, and to ignore the `ok` status of the function.) The function takes some global state or context as input and invokes three functions in turn, each of which takes a context value and returns both some data and a new context. The context is effectively *threaded through* the program. This allows us to have a mutable state in a programming language without side effects.

The above approach is error-prone—it is all too easy to forget one prime (') and pass the wrong context to a function—as well as cluttered by all the context variables. What if we could simply write the following instead?

```
def welcome_new_user₂ (user_name : string) (ctxt : context) :
  (ℕ × string) × context :=
do
  user ← create_user user_name,
  password ← generate_temporary_password user,
  ok ← send_unencrypted_email user password,
  pure (user, password)
```

This is what the state monad provides.

The underlying type constructor captures the concept of computations or actions over states of type σ and with return values of type α:[1]

[1] This type constructor is traditionally called `State` or `state`, but these names are awfully confusing. In this case, it seems preferable to honor tradition in the breach rather than in the observance.

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

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

A stateful action is a function that takes some state and returns both a value and some new state. The `σ →` part of `action`'s definition provides the old state; the left component of the cartesian product, α, provides the result of the computation; and the right component of the product, σ, provides the new state. Thus, the state is implicitly threaded through the program. As with other effectful programs, the `do` notation only exposes the data—the value of type α—and conceals the effect—the old and new σ states.

The identity monad corresponds to the case where `σ := unit`, a type of cardinality one (similar to `void` in Java or C) whose unique value is written `()`. The type `unit → α × unit` is isomorphic to α. This intuition can guide us when defining `pure` and `bind`.

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

```
def action.read {σ : Type} : action σ σ
| s := (s, s)

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

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

def action.bind {σ : Type} {α β : Type} (ma : action σ α)
    (f : α → action σ β) :
  action σ β
| s :=
  match ma s with
  | (a, s') := f a s'
  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 `()`. 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 exploit functional extensionality (`funext`): To prove `f = g`, it suffices to show `∀x, f x = g x`.

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

As a concrete example, the following program removes all elements that are smaller than a previous element in the list, leaving us with a list of increasing elements. The maximal element is stored as the state σ.

```
def diff_list : list ℕ → action ℕ (list ℕ)
| []        := pure []
| (n :: ns) :=
  do
    prev ← action.read,
    if n < prev then
      diff_list ns
    else
      do
        action.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 an initial state. The last state is returned along with the resulting list. It corresponds to the largest element seen in the list or the start state. Thus, the commands

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

produce the output

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

6.7 Nondeterminism

Whereas the option monad stores zero or one α values and the identity and state monads store exactly one α value, the *set monad* provides an arbitrary, possibly infinite number of α values. This is useful to model nondeterminism, as sets of possible behaviors.

Lean's type `set α` is defined as `α → Prop`. In other words, a set is identified with its characteristic predicate. Familiar operators such as the empty set (∅), the universal set (`set.univ`), union (∪), intersection (∩), and membership (∈) are supported, as well as traditional curly brace notations such as `{a}`, `{a, b}`, and `{x | p x}`. Many set constructs can be simplified by `simp`.

The `set` type constructor can be registered as a lawful monad as follows:

```
def set.pure {α : Type} : α → set α
| a := {a}

def set.bind {α β : Type} : set α → (α → set β) → set β
| A f := {b | ∃a, a ∈ A ∧ b ∈ f a}

@[instance] def set.lawful_monad : lawful_monad set :=
{ pure       := @set.pure,
  bind       := @set.bind,
  pure_bind  :=
    begin
      intros α β a f,
      simp [set.pure, set.bind]
    end,
  bind_pure  :=
    begin
      intros α m,
      simp [set.pure, set.bind]
    end,
  bind_assoc :=
    begin
      intros α β γ f g m,
      simp [set.pure, set.bind],
      apply set.ext,
      simp,
      tautology
    end }
```

The `pure` operation simply puts the given value `a` in a singleton set `{a}`. The `bind` operation calls `f` on all values in the set `A` and returns the union of all the results. For example, if `A := {3, 8}` and `f := (λa, {a, a + 1})`, then `set.bind A f` equals `{3, 4, 8, 9}`. The last proof relies on *set extensionality*, which states that two sets

that contain the same elements must be equal:

```
set.ext {α : Type} {A B : set α} : (∀x, x ∈ A ↔ x ∈ B) → A = B
```

Another noteworthy point is the use of the `tautology` tactic. The goal's target is `∀x, (∃a, (∃a_1, a_1 ∈ m ∧ a ∈ f a_1) ∧ x ∈ g a) ↔ (∃a, a ∈ m ∧ ∃a_1, a_1 ∈ f a ∧ x ∈ g a_1)`, where the two sides of `↔` are the same except for the placement of the existential quantifiers (and, confusingly, the names of the bound variables). This ugly proposition could be proved by a tedious sequence of introduction and elimination, but we deserve more automation.

6.8 Tautology Tactic

tautology

The `tautology` tactic performs elimination of the logical symbols `∧`, `∨`, `↔`, and `∃` in hypotheses and introduction of `∧`, `↔`, and `∃` in the target, until all the emerging subgoals can be trivially proved (e.g., by `refl`).

6.9 A Generic Algorithm: Iteration over a List

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

```
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 `m` value containing a list and not a list of `m` values. Try to work out why it is well typed and has the desired behavior.

The `mmap` function distributes over the append operator `++`. The `do` 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, lawful_monad.bind_pure, lawful_monad.pure_bind]
| (a :: as) as' :=
  by simp [mmap, mmap_append as as', lawful_monad.pure_bind,
    lawful_monad.bind_assoc]
```

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` extracts the `(n + 1)`st element of each list in `xss`. It fails with `option.none` if one of the list is not long enough. Running

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

returns `option.some [13, 23, 33]`.

Lean already includes a function called `list.mmap` corresponding to our `mmap`.

6.10 Summary of New Constructs

Notations

`do`	indicates the start of a monadic program
`let`	introduces a local definition in a monad
`<|>`	tries the left-hand side first; tries the right-hand side on failure
`>>=`	composes monadic computations

Lemma

`set.ext`	set extensionality

Tactic

`tautology`	proves tautologies involving the basic logical symbols

Chapter 7

Metaprogramming

Like most proof assistants, Lean can be extended with custom tactics and other functionality. This kind of programming—programming the proof assistant—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. Monads are used to provide access to the proof assistant's state and handle tactic failures.

Abstract syntax trees, presented as inductive types in Lean, *reflect* internal data structures. In a metaprogram, Lean terms are viewed as an inductive type with ten constructors, but internally they are stored in a C++ data structure. The proof assistant's C++ internals are exposed through Lean interfaces, which we can use for accessing the current goal, unifying terms, querying and modifying the environment, and setting attributes (e.g., `@[simp]`).

Here are some example applications of metaprogramming:

- goal transformations (e.g., applying safe introduction rules, put the goal in negation normal form);
- heuristic proof search (e.g., applying unsafe introduction rules with backtracking);
- decision procedures (e.g., for linear arithmetic, propositional logic);
- definition generators (e.g., Haskell-style `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).

As mathematician and Lean user Kevin Buzzard wrote:[1]

> If you find yourself "grinding" (to use a computer game phrase), doing the same sort of stuff over and over again because you need to do it to make progress, then you can try to persuade a computer scientist to write a tactic to do it for you (or even write your own tactic if you're brave enough to write `meta` Lean code).

This chapter is inspired by an article by Ebner et al. [5] and by Chapter 8 of *Programming in Lean* [2].

[1] `https://xenaproject.wordpress.com/2020/02/09/lean-is-better-for-proper-maths-than-all-the-other-theorem-provers/`

7.1 Tactics and Tactic Combinators

When programming our own tactics, we often need to repeat some actions on several goals, or to recover if a tactic fails. Tactic combinators help in such case. The most important one is `repeat { tactic }`. It applies the tactic specified within curly braces repeatedly on all goals, emerging subgoals, emerging subsubgoals, and so on, until the tactic cannot be applied any further. Here is an example of `repeat` in action:

```
lemma repeat_example :
  even 4 ∧ even 7 ∧ even 3 ∧ even 0 :=
begin
  repeat { apply and.intro },
  repeat { apply even.add_two },
```

After the first `repeat`, the proof state consists of four goals:

```
⊢ even 4        ⊢ even 7        ⊢ even 3        ⊢ even 0
```

There are no more conjuctions in sight. The second `repeat`, which applies the lemma `even.add_two : ∀k, even k → even (k + 2)` over and over, leaves us with these goals:

```
⊢ even 0        ⊢ even 1        ⊢ even 1        ⊢ even 0
```

The first and last goals are annoying because they correspond to the lemma `even.zero`. We can prove them by trying to apply `even.zero` whenever applying `even.add_two` fails. This is achieved as follows:

```
lemma repeat_orelse_example :
  even 4 ∧ even 7 ∧ even 3 ∧ even 0 :=
begin
  repeat { apply and.intro },
  repeat {
    apply even.add_two
    <|> apply even.zero },
```

The tactic combinator `<|>` first executes its left-hand side. If this fails, the right-hand side is executed instead. If the right-hand side also fails, the entire combinator fails. With the new tactic, we have two unprovable goals left:

```
⊢ even 1                ⊢ even 1
```

The `repeat` combinator is very aggressive, targeting all goals and subgoals recursively. A less ambitious variant is `iterate`. It works repeatedly on the first goal until it fails; then it stops. Consider this example:

```
lemma iterate_orelse_example :
  even 4 ∧ even 7 ∧ even 3 ∧ even 0 :=
begin
  repeat { apply and.intro },
  iterate {
    apply even.add_two
    <|> apply even.zero },
```

After three applications of `even.add_two` or `even.zero`, the first goal cedes its place to the second goal. The `iterate` combinator keeps on working on this goal, until it gets stuck:

```
⊢ even 1        ⊢ even 3        ⊢ even 0
```

Notice that the last two goals are never considered.

The next combinator, `all_goals`, applies a tactic exactly once to each goal. The combinator succeeds only if the inner tactic succeeds on *all* goals. For example, it fails below, because `even.add_two` cannot be applied on the goal `⊢ even 0`:

```
lemma all_goals_example :
  even 4 ∧ even 7 ∧ even 3 ∧ even 0 :=
begin
  repeat { apply and.intro },
  all_goals { apply even.add_two },   -- fails
```

To ignore failures of the inner tactic, we can wrap it in the `try` combinator:

```
lemma all_goals_try_example :
  even 4 ∧ even 7 ∧ even 3 ∧ even 0 :=
begin
  repeat { apply and.intro },
  all_goals { try { apply even.add_two } },
```

The resulting state is

```
⊢ even 2        ⊢ even 5        ⊢ even 1        ⊢ even 0
```

The construct `try { `*`tactic`*` }` is equivalent to *`tactic`* `<|> skip`, where `skip` is a tactic that succeeds without doing anything. A related tactic is `done`; it succeeds if there are no goals left and fails otherwise.

Yet another variant is the `any_goals` combinator. Like `all_goals`, it applies a tactic exactly once to each goal. But unlike `all_goals`, `any_goals` succeeds if the inner tactic succeeds on *any* goal. The example

```
lemma any_goals_example :
  even 4 ∧ even 7 ∧ even 3 ∧ even 0 :=
begin
  repeat { apply and.intro },
  any_goals { apply even.add_two },
```

results in the state

```
⊢ even 2        ⊢ even 5        ⊢ even 1        ⊢ even 0
```

This is the same state as in the previous example. The difference is that `any_goals` can fail whereas `all_goals { try { ... } }` always succeeds.

Sometimes we want to leave a goal alone unless we can fully prove it. The `solve1` combinator takes an arbitrary tactic as argument and transforms it into an all-or-nothing tactic (sometimes called a terminal or end-game tactic). If the inner tactic does not prove the goal, the combinator fails. In the example

```
lemma any_goals_solve1_repeat_orelse_example :
  even 4 ∧ even 7 ∧ even 3 ∧ even 0 :=
begin
  repeat { apply and.intro },
  any_goals { solve1 { repeat {
    apply even.add_two
    <|> apply even.zero } } },
```

the first and fourth goals are proved, and we are left with the two unprovable goals, exactly as they stand in the lemma statement:

```
⊢ even 7                    ⊢ even 3
```

The combinators `repeat`, `iterate`, `all_goals`, and `any_goals` can lead to infinite looping. Consider this example:

```
lemma repeat_not_example :
  ¬ even 1 :=
begin
  repeat { apply not.intro },
```

The `not.intro` rule is `(?a → false) → ¬ ?a`, so it applies once to transform the goal into `⊢ even 1 → false`. Because `¬ ?a` is defined as `?a → false`, the rule applies again, yielding the same goal again. This process can be repeated forever.

So far we have not left the comfort of Lean's interactive interface level. It is time to start with the actual metaprogramming, by coding a custom tactic. The tactic embodies the behavior we hardcoded in the `solve1` example above:

```
meta def intro_and_even : tactic unit :=
do
  tactic.repeat (tactic.applyc ``and.intro),
  tactic.any_goals (tactic.solve1 (tactic.repeat
    (tactic.applyc ``even.add_two
     <|> tactic.applyc ``even.zero)))
```

Let us analyze this code step by step:

- The first line starts with `meta def`, indicating the declaration of a metafunction. These are much like regular functions declared without the `meta` keyword, but they do not need to pass the termination check.
- The first line declares a metafunction called `intro_and_even` of type `tactic unit`, where `tactic α` is the type of metaprograms returning a value of type `α`. (Since tactics are the main kind of metaprograms, metaprograms are often called "tactics" in Lean.) Often, metaprograms only have side effects on the state but return no value. In such cases, they can return the empty tuple `()`, of type `unit`.
- The second line, `do`, marks the entry into a monad: the metaprogramming monad (`tactic`).
- The third line is the low-level, metaprogram equivalent of the familiar tactic invocation `repeat { apply and.intro }`. The `tactic.repeat` metafunction corresponds to `repeat`. The `tactic.applyc` metafunction is one of several that correspond to `apply`. It applies a lemma specified by its name, of type

`name`. The name here is ``` ``and.intro ```. The two backticks resolve the name of a Lean constant and create a value of type `name`.

- The rest of the metaprogram follows the same principles. Notice how the monad operator `<|>` is used to provide the same effect as the tactic combinator `<|>`. The metaprogramming monad is an alternative monad.

Tactics and tactic combinators often have the same name at the user interface level and at the metaprogramming level, except for the `tactic.` prefix. A notable exception is `sorry`, which is called `tactic.admit`.

Once we have defined a custom tactic, we can apply it as usual:

```
lemma any_goals_solve1_repeat_orelse_example₂ :
  even 4 ∧ even 7 ∧ even 3 ∧ even 0 :=
begin
  intro_and_even,
```

This yields the subgoals

```
⊢ even 7                    ⊢ even 3
```

A key benefit of Lean's metaprogramming framework is that we do not need to learn another programming language to write metaprograms. We can work with the same constructs and notation used to define ordinary objects in Lean's library. Everything in that library is available for metaprogramming purposes (e.g., $\mathbb{Z}$, `list`). 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. All we need is some basic understanding of monads and some familiarity with Lean's metaprogramming interfaces.

We conclude this section by introducing a further tactic combinator that can be useful for metaprogramming. The "and then" operator `;` can be used to join two tactics. The left-hand side is executed on the first goal. The right-hand side is executed on every emerging subgoals (but not on the original second goal, third goal, etc.). Like `<|>`, the `;` operator has the same syntax at the interactive interface level and at the metaprogramming level. Thus, we can write

```
by induction n; simp [*]
```

instead of

```
begin
  induction n,
  { simp },
  { simp [n_ih] }
end
```

7.2 The Metaprogramming Monad

The metaprogramming monad, `tactic`, 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]` lemmas).

- It also behaves like an option monad. The program `tactic.fail` *`string`*, where *`string`* is an error message, leaves the monad in an error state.
- It is an alternative monad. Namely, it provides an operator <|> for specifying alternatives.
- It provides some tracing capabilities. We can use `tactic.trace` *`string`*, or we can display the current proof state using `tactic.trace_state`.

In addition, metaprograms have access to the current goals. Each goal is represented as a metavariable `?m` standing for a missing proof term. Each metavariable has a type and a local context specifying the variables and hypotheses that can be used to instantiate `?m`. By the Curry–Howard correspondence, variables and hypotheses are not distinguished. Metaprograms have access to Lean's functions for inferring implicit arguments and types.

Let us put the metaprogramming monad to some use:

```
lemma even_14 :
  even 14 :=
by do
  tactic.trace "Proving evenness ...",
  intro_and_even
```

This first example is very modest: It uses the monad's tracing facilities to print the message "Proving evenness ..." before applying the `intro_and_even` tactic we developed above. The `do` keyword signals entry into the monad. The `by` keyword preceding it is necessary to enter tactic mode, as usual.

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 apply it to prove that 16 is even:

```
meta def hello_then_intro_and_even : tactic unit :=
do
  tactic.trace "Proving evenness ...",
  intro_and_even

lemma even_16 :
  even 16 :=
by hello_then_intro_and_even
```

Notice, again, the presence of the `meta` keyword in front of the function definition. Any executable 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, because they might be logically inconsistent. Metaprograms can also call other metaprograms, some of which are implemented in C++.

Most metaprograms are designed to work on a proof state, as a tactic, but some can also be used on the top level, without a proof context, by invoking the `run_cmd` command:

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

To adapt to the situation in which they are invoked, tactics must be able to inspect the current goal. The following example outputs the local context, the target, and all goals (both the metavariables and their types):

```
meta def trace_goals : tactic unit :=
do
  tactic.trace "local context:",
  ctx ← tactic.local_context,
  tactic.trace ctx,
  tactic.trace "target:",
  P ← tactic.target,
  tactic.trace P,
  tactic.trace "all missing proofs:",
  Hs ← tactic.get_goals,
  tactic.trace Hs,
  τs ← list.mmap tactic.infer_type Hs,
  tactic.trace τs

lemma even_18_and_even_20 (α : Type) (a : α) :
  even 18 ∧ even 20 :=
by do
  tactic.applyc ``and.intro,
  trace_goals,
  intro_and_even
```

The output, which is visible by hovering over the `do` keyword, is as follows:

```
local context:
[α, a]
target:
even 18
all missing proofs:
[?m_1, ?m_1]
[even 18, even 20]
```

The goals are a list of metavariables that must be instantiated. They are normally not displayed in the proof state; instead, only their types are shown. In the example, the goals are `?m_1 : even 18` and `?m_1 : even 20`. Confusingly, Lean reuses the same metavariable names for different goals.

The metafunctions in the above programs have the following types:

```
tactic.trace : α → tactic unit
tactic.local_context : tactic (list expr)
tactic.target : tactic expr
tactic.get_goals : tactic (list expr)
tactic.infer_type : expr → tactic expr
```

where `expr` stores a term and `α` is restricted by a type class (not shown) to be printable. In addition, `list.mmap` (Section 6.9) is typed as follows:

```
list.mmap {α β : Type} : (α → m β) → list α → m (list β)
```

In practice, instead of the ad hoc `trace_goals` tactic, we would normally use the following predefined tactic:

```
tactic.trace_state : tactic unit
```

The next example is more realistic. It implements a `hypothesis` tactic that, like the predefined `assumption` tactic, looks for a hypothesis of the right type (i.e., the right proposition) and applies it to prove the current goal:

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

meta def hypothesis : tactic unit :=
do
  hs ← tactic.local_context,
  exact_list hs
```

The main metafunction 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. 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 `tactic.fail`.

A simple invocation follows:

```
lemma p_a_of_p_a {α : Type} {p : α → Prop} {a : α} (h : p a) :
  p a :=
by hypothesis
```

The tracing informs us that `α`, `p`, `a`, and `h` were tried in turn before the matching `h` was found and successfully applied.

The example featured the metafunction

```
tactic.fail {α β : Type} : α → tactic β
```

where `α` must support printing.

7.3 Names, Expressions, Declarations, and Environments

The metaprogramming framework is articulated around five main types:

- `tactic` manages the proof state, the global context, and more;
- `name` represents a structured name (e.g., `x`, `even.add_two`);
- `expr` represents an expression—i.e., a term, type, proof, or proposition—as an abstract syntax tree;
- `declaration` represents a constant declaration, a definition, an axiom, or a lemma;
- `environment` stores all the declarations and notations that make up the global context.

The type `expr` of expressions is presented as follows to metaprograms:

```
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
```

Let us briefly review the constructors in turn:

- `expr.var` represents a bound variable, using a notation known as De Bruijn indices. For example, `expr.var 0` refers to the variable bound by the closest λ- or ∀-binder, `expr.var 1` refers to the variable bound by the second closest binder; and so on.
- `expr.sort` is used to represent the types of types. For example, `expr.sort level.zero` represents `Prop`, and `expr.sort (level.succ level.zero)` represents `Type`.
- `expr.const` represents a constant, such as `list.reverse` or ℕ. The second argument represents universe levels, a concept that will be explained in Chapter 11.
- `expr.mvar` represents a metavariable, i.e., a variable `?m` with a question mark. The first `name` argument is a unique name for the metavariable; the second `name` is a "pretty" name; and the `expr` either stores the metavariable's type or is a dummy value.
- `expr.local_const` represents a variable in the local context (e.g., `x`, `h`). The first `name` argument is a unique name for the variable; the second `name` is a "pretty" name; the `binder_info` stores whether the variable is an explicit `( )`, implicit `{ }`, or type class `[ ]` argument; and the `expr` either stores the variable's type or is a dummy value.
- `expr.app` represents the application of a function to an argument.
- `expr.lam` represents a λ-expression. The `name` is the bound variable's name, the `binder_info` is as for `expr.local_const`; the first `expr` is the bound variable's type; and the second `expr` is the λ-expression's body.
- `expr.pi` represents a ∀-quantifier, including the degenerate case where the bound variable does not occur in the body (→). The arguments are as for `expr.lam`.
- `expr.elet` represents a `let` expression. The `name` is the name of the definition variable; the first `expr` is its type; the second `expr` is the right-hand side of the definition; and the third `expr` is the `let` expression's body.
- `expr.macro` is a low-level mechanism used to implement `sorry`, nameless placeholders (_), and recursive metaprograms.

To each constructor corresponds a discriminator of type `expr → bool`. For example, `expr.is_pi e` returns `tt` if `e` is of the form `expr.pi ...`; otherwise, it re-

turns `ff`. The names are easy to guess, with a few exceptions: `expr.is_constant`, `expr.is_local_constant`, `expr.is_lambda`, and `expr.is_let`.

A lightweight way to construct an expression uses 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—‘(*expr*)—must be fully elaborated. In other words, it may not contain placeholders, as in the second example:

```
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
  tactic.trace e
```

With two backticks—‘‘(*expr*)—we have *pre-expressions*. These are Lean expressions that may contain some placeholders to be filled in later:

```
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
```

The type `pexpr` is a variant of `expr`. A `pexpr` can be elaborated into an expression using `tactic.to_expr`, which has type `pexpr → tactic expr`.

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

```
run_cmd do
  let e : pexpr := ‘‘‘(seattle.washington),
  tactic.trace e
```

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

```
run_cmd tactic.trace ‘and.intro
run_cmd tactic.trace ‘intro_and_even
run_cmd tactic.trace ‘seattle.washington

run_cmd tactic.trace ‘‘and.intro
run_cmd tactic.trace ‘‘intro_and_even
run_cmd tactic.trace ‘‘seattle.washington   -- fails
```

The single-backtick variant—‘*name*—does not check whether the name exists. The two-backtick variant—‘‘*name*—fully elaborates the name of a constant and returns an error if no constant of that name exists in the current global context. Whether the check and elaboration is desirable depends on the situation. In the demonstration file associated with this chapter, ‘`intro_and_even` returns the name `intro_and_even` (of the metafunction we defined in Section 7.1), whereas ‘‘`intro_and_even` expands to the fully elaborated name `LoVe.intro_and_even`,

including the namespace `LoVe`. Tactics such as `tactic.applyc` expect a fully elaborated name. This explains why we wrote `tactic.applyc ``and.intro` earlier. A name can be converted to a string using the polymorphic `to_string` operation.

Sometimes we would like to embed an existing expression in a larger expression. This can be achieved using *antiquotations*. These are announced by the 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 : pexpr := ``(list.map %%x),
  tactic.trace e

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

The three commands print `2 + 1`, `list.map (as_is (id.{1} nat))`, and `a 6._.13 (as_is (id.{1} nat))`, respectively. Admittedly, this is rather cryptic.

Antiquotations can also be used to perform pattern matching:

```
lemma one_add_two_eq_three :
  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)
```

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`. This also succeeds, binding `α`, `l`, `r` to `ℕ`, `1 + 2`, and `3`, respectively.

Declarations are presented as an inductive type to metaprogrammers:

```
meta inductive declaration
| defn : name → list name → expr → expr →
  reducibility_hints → bool → declaration
| thm  : name → list name → expr → task expr → declaration
```

```
| cnst : name → list name → expr → bool → declaration
| ax   : name → list name → expr → declaration
```

The constructor arguments are documented in Lean's source code. For example, in a declaration of the form `declaration.defn nam _ τ rhs _ _`, `nam` is the name of the defined constant, `τ` is its type, and `rhs` is the definition's body. Given a declaration, `declaration.to_name` extracts its first component, of type `name`.

The `environment` type is presented as an abstract type, equipped with some operations to query and modify it. The most useful metafunction is

```
environment.fold {α : Type} :
  environment → α → (declaration → α → α) → α
```

A call to `environment.fold env x f` applies the given function `f` on each declaration in the environment in turn. In addition, an accumulator value is threaded through. Its initial value is `x` and the final value is returned at the end. For example, the following command outputs the number of declarations in the environment:

```
run_cmd do
  env ← tactic.get_env,
  tactic.trace (environment.fold env 0 (λdecl n, n + 1))
```

Fold functions are a common idiom in functional programming. If you are not familiar with the concept, we recommend the chapter "Higher Order Functions" of *Learn You a Haskell for Great Good!* [19]. You can start reading from the heading "Only folds and horses."

7.4 First Example: A Conjunction-Destructing Tactic

In this and the next section, we review two metaprograms that accomplish well-defined tasks. The first program is a tactic called `destruct_and` that automates the elimination of conjunctions in premises. Our aim is to automate proofs such as the following:

```
lemma abcd_a (a b c d : Prop) (h : a ∧ (b ∧ c) ∧ d) :
  a :=
and.elim_left h

lemma abcd_b (a b c d : Prop) (h : a ∧ (b ∧ c) ∧ d) :
  b :=
and.elim_left (and.elim_left (and.elim_right h))

lemma abcd_bc (a b c d : Prop) (h : a ∧ (b ∧ c) ∧ d) :
  b ∧ c :=
and.elim_left (and.elim_right h)
```

In each case, we would like to simply write `destruct_and h`.

Our tactic relies on a helper metafunction, which takes as argument the hypothesis `h` to use as an expression rather than as a name:

```
meta def destruct_and_helper : expr → tactic unit
| h :=
  do
```

```
  t ← tactic.infer_type h,
  match t with
  | `(%%a ∧ %%b) :=
    tactic.exact h
    <|>
    do {
      ha ← tactic.to_expr ``(and.elim_left %%h),
      destruct_and_helper ha }
    <|>
    do {
      hb ← tactic.to_expr ``(and.elim_right %%h),
      destruct_and_helper hb }
  | _            := tactic.exact h
  end
```

We first extract `h`'s type (i.e., its proposition). We perform pattern matching with antiquotations to see if the type is of the form `a ∧ b`. If so, we try three things in turn. First, we invoke `exact h`. If this fails, we apply the left elimination rule on `h`, producing the proof term `and.elim_left h : a`, and continue recursively with that proof term. The `tactic.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 metafunction has very little to do:

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

The metafunction retrieves the hypothesis `h` with the specified name `nam` using `tactic.get_local`, which is of type `name → tactic expr`, and calls the helper metafunction.

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

```
lemma abc_a (a b c : Prop) (h : a ∧ b ∧ c) :
  a :=
by destruct_and `h

lemma abc_b (a b c : Prop) (h : a ∧ b ∧ c) :
  b :=
by destruct_and `h

lemma abc_bc (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 Second Example: A Provability Advisor

It sometimes happens that we state a lemma, prove it, and later realize that the lemma already existed in the library. To help prevent this, we implement a `prove_direct` tool that traverses all lemmas in the database and checks whether one of them can be used to prove the current goal. A similar tactic is available in `mathlib` under the name `library_search`.

We will review the code in steps. The first step is a metafunction `is_theorem` that returns `tt` if a declaration is a lemma or an axiom and `ff` otherwise:

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

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

The next metafunction returns the list of all theorem 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 `tactic.get_env` metafunction returns an `environment` object representing the global context. 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 `is_theorem`. At the end, `environment.fold` returns the accumulator.

Another component we need is a tactic that attempts to prove the goal using a theorem specified by its name:

```
meta def prove_with_name (nam : name) : tactic unit :=
do
  tactic.applyc nam
    ({ md := tactic.transparency.reducible, unify := ff }
     : tactic.apply_cfg),
  tactic.all_goals tactic.assumption
```

The `tactic.applyc` applies the theorem named `nam` to the current goal. For performance, we pass an optional configuration, `{ md := transparency.reducible, unify := ff }`, which tells `tactic.applyc` to perform less computational expansion. This is important since this tactic will be invoked thousands of times. The `tactic.all_goals` combinator, together with `tactic.assumption`, is used to ensure that the hypothesis from the local context are used to prove all remaining subgoals.

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

```
meta def prove_direct : tactic unit :=
do
```

```
    nams ← get_all_theorems,
    list.mfirst (λnam,
        do
          prove_with_name nam,
          tactic.trace ("directly proved by " ++ to_string nam))
      nams
```

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

As a small refinement of the above, we propose a version of `prove_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 prove_direct_symm : tactic unit :=
prove_direct
<|>
do {
  tactic.applyc `eq.symm,
  prove_direct }
```

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

7.6 A Look at Two Predefined Tactics

Quite a few of Lean's predefined tactics are implemented as metaprograms and not in C++. We can find these definitions by clicking the name of a construct in Visual Studio Code while holding the control or command key. To familiarize ourselves further with the metaprogramming framework, we conclude this chapter by looking at the implementation of two familiar tactics that were introduced in Chapter 2: `intro` and `assumption`.

We can find `intro`'s code by entering `#check tactic.intro` and by clicking `tactic.intro` while holding control or command:

```
meta def intro (n : name) : tactic expr :=
do
  t ← target,
  if expr.is_pi t ∨ expr.is_let t then
    intro_core n
  else
    whnf_target >> intro_core n
```

The definition is located inside the `tactic` namespace, explaining why `tactic.intro` and `tactic.target` are simply called `intro` and `target`. If the goal's target is a ∀-quantifier or a `let` expression, `tactic.intro_core` is invoked. Otherwise, a tactic named `tactic.whnf_target` is invoked before `tactic.intro_core`, in the hope that a ∀ or `let` will emerge. The `>>` operator performs sequential composition in a monad; `ma >> mb` is syntactic sugar for `do ma, mb`.

If we follow the definition of `tactic.intro_core`, we reach a dead end:

```
meta constant intro_core : name → tactic expr
```

The metafunction is simply declared as a constant, without a definition. This indicates that the implementation is in C++. The core of the `whnf_target` tactic, which normalizes the goal by expanding definitions, performing β-reductions, and more, is also implemented in C++, for efficiency reasons.

In Section 7.2, we reviewed the implementation of a `hypothesis` tactic that provided essentially the same functionality as `assumption`. Let us now see how `assumption` is actually implemented:

```
meta def assumption : tactic unit :=
do {
  ctx ← local_context,
  t ← target,
  H ← find_same_type t ctx,
  exact H }
<|> fail "assumption tactic failed"
```

Again, the definition is located in the `tactic` namespace. The `assumption` tactic attempts to find a hypothesis `H` (of type `expr`) among the variables declared in the local context `ctx` that has a type unifiable with that of the target `t` (i.e., a hypothesis whose statement is unifiable with `t`). Then `H` is applied to the target using `tactic.exact`. If anything goes wrong, the `assumption` tactic catches the error and reports "assumption tactic failed."

The auxiliary metafunction `find_same_type` is defined as follows:

```
meta def find_same_type : expr → list expr → tactic expr
| e []        := failed
| e (H :: Hs) :=
  do
    t ← infer_type H,
    (unify e t >> return H)
    <|> find_same_type e Hs
```

The metafunction takes an expression `e` representing a type (i.e., proposition) and a context and returns an expression. If the context is empty, a message-less error is raised. Otherwise, we check if the local context's first hypothesis `H`'s type `t` is unifiable with `e`. If so, `H` is returned; otherwise, we continue with the remaining hypotheses `Hs`. The `return` operation is an alias for `pure`.

The `find_same_type` metafunction relies on two general-purpose metafunctions we had not encountered before:

```
tactic.failed {α : Type} : tactic α
tactic.unify : expr → expr → tactic unit
```

7.7 Miscellaneous Tactics

skip

The `skip` tactic succeeds without doing anything. It is sometimes useful when writing custom tactics.

done

The `done` tactic raises a failure if there are some goals left; otherwise, it succeeds without doing anything. It is sometimes useful when writing custom tactics.

library_search

The `library_search` tactic searches the loaded libraries for a lemma that exactly proves the goal. On success, it suggests a tactic invocation of the form `exact ...` which can be inserted in the formalization.

7.8 Summary of New Constructs

Command

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

Declaration

`meta`	prefixes declarations of metaprograms

Quotations

‘*n*	quotes literal name, with checking
‘‘*n*	quotes literal name, without checking
‘(*e*)	quotes fully elaborated expressions
‘‘(*e*)	quotes pre-expressions, with name checking
‘‘‘(*e*)	quotes pre-expressions, without name checking
%%*x*	embeds an expression within a quotation

Tactic

`done`	fails if there are some goals left
`library_search`	searches the libraries for a lemma that proves the current goal
`skip`	does nothing

Tactic Combinators

`;`	applies the second tactic to all subgoals yielded by the first tactic
`<|>`	tries the left-hand side first; tries the right-hand side on failure
`all_goals`	applies a tactic once on each goal, expecting only successes
`any_goals`	applies a tactic once on each goal, expecting at least one success
`iterate`	repeatedly applies a tactic on the first goal until failure
`repeat`	repeatedly applies a tactic on all goals and subgoals until failure
`solve1`	transforms a tactic into an all-or-nothing tactic
`try`	tries to apply a tactic; does nothing on failure

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, to prove properties of the semantics, and to reason about concrete programs.

This chapter is inspired by Chapter 7 of *Concrete Semantics: With Isabelle/HOL* [24].

8.1 Formal Semantics

A formal semantics allows us to specify and reason about a programming language and about individual programs written in that language. It can form the basis of verified compilers, interpreters, verifiers, static analyzers, type checkers, and more. Without formal proofs, these tools are almost always wrong.

Consider WebAssembly. It is a new machine-like language for web browsers, designed as a portable target for compiling high-level languages such as C++ and Rust. A researcher, Conrad Watt [31], formalized its semantics and type system using the Isabelle/HOL proof assistant. Here is what he had to report (our italics):

> We have produced a full Isabelle mechanisation of the core execution semantics and type system of the WebAssembly language. In addition, we have created a mechanised proof for the type soundness properties stated in the working group's paper. In order to complete this proof, *several deficiencies* in the official WebAssembly specification, uncovered by our proof and modelling work, needed to be corrected by the specification authors. In some cases, these meant that *the type system was originally unsound*.
>
> We have maintained a constructive dialogue with elements of the working group, mechanising and verifying new features as they are added to the specification. In particular, the mechanism by which a WebAssembly implementation interfaces with its host environment was not formally specified in the working group's original paper. Extending our mechanisation to model this feature *revealed a deficiency* in the WebAssembly specification that *sabotaged the soundness of the type system*.

Watt's research is only one example among many. Proof assistants are widely used for programming language research. Every year, about 10%–20% of papers

presented at the Principles of Programming Languages (POPL) conference are formalized. This is possible because comparatively little machinery is needed to get started. The proofs tend to have lots of cases, which is a good match for computers. Moreover, proof assistants are extremely convenient to keep track of what needs to be changed as we extend a programming language with more features.

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` S	(`while` loop)

where S stands for a *statement* (also called *command* or *program*), x for a program variable, a for an arithmetic expression, and b for a Boolean expression.

In our grammar, we deliberately leave the syntax of arithmetic and Boolean expressions unspecified. In Lean, we have the choice:

- We can use a type such as `aexp` from Section 1.2 and similarly for Booleans.
- We can decide that an arithmetic expression is simply a function from states to natural numbers (e.g., `state` $\to \mathbb{N}$) and a Boolean expression is simply a predicate over states (e.g., `state` $\to$ `bool` or `state` $\to$ `Prop`). A *state* is a mapping from program variables to values. Thus, `x + y + 1` would be represented by the function `λs : state, s "x" + s "y" + 1`, and `a` $\neq$ `b` would be represented by the predicate `λs : state, s "a"` $\neq$ `s "b"`.

These two options correspond to the difference between deep and shallow embeddings. A *deep embedding* of some syntax (expression, formula, program, etc.) consists of an abstract syntax tree specified in the proof assistant (e.g., `aexp`) with a semantics (e.g., `eval`). In contrast, a *shallow embedding* simply reuses the corresponding mechanisms from the logic (e.g., functions and predicates).

A deep embedding allows us to reason about a program's syntax. A shallow embedding is more lightweight, because we can use it directly, without having to define a semantics. A shallow embedding is its own semantics.

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

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

infixr ‘ ;; ‘ : 90 := stmt.seq
```

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

The infix syntax `S ;; T` is provided as an alternative to `stmt.seq S T`. We cannot use a single semicolon `;` because it already means "and then" (Section 7.1).

The correspondence between the inductive type's constructors and the WHILE grammar rules should be clear. Variables are represented by strings. The type `state` is defined as `string` $\to \mathbb{N}$, 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 interpreter. There are two main variants: big-step semantics and small-step semantics. We will start by giving a big-step semantics to our WHILE language.

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

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

For deterministic languages, the phrase "may terminate" is equivalent to "must terminate" or simply "terminates."

In keeping with the definition of WHILE programs, a state `s` is a function of type `string` $\to \mathbb{N}$. An example of a judgment follows:

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

We use the informal notation `[x` $\mapsto$ `3, y` $\mapsto$ `5]` to represent the function `λv, if v = "x" then 3 else if v = "y" then 5 else 0` and similarly for `[x` $\mapsto$ `8, y` $\mapsto$ `0]`. Intuitively, the judgment should hold.

The traditional way to specify such a semantics is through a formal system of derivation rules, in the style of the typing rules presented in Sections 1.1.3 and 3.6. The derivation rules for big-step semantics judgments are given below. The rules can be seen as an idealized interpreter for WHILE programs.

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

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

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

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

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

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

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

In the rules, `a s` denotes the value of arithmetic expression `a` in state `s`, and similarly for `b s`. Moreover, the syntax `s{x ↦ n}` represents the state that is identical to `s` except that it maps the variable `x` to `n`. Formally:

```
s{x ↦ n} = (λv, if v = x then n else s v)
```

This syntax and its automation are provided by the LoVelib library.

As an exercise, let us derive the example judgment above. Let `s := [x ↦ 3, y ↦ 5]`, `t := [x ↦ 8, y ↦ 5]`, and `u := [x ↦ 8, y ↦ 0]`. Then we have

$$\dfrac{\dfrac{}{(x := x + y,\ s) \Longrightarrow t}\ \textsc{Asn} \qquad \dfrac{}{(y := 0,\ t) \Longrightarrow u}\ \textsc{Asn}}{(x := x + y;\ y := 0,\ s) \Longrightarrow u}\ \textsc{Seq}$$

The derivation rules can be read intuitively. Consider Seq:

> If (1) executing `S` in state `s` leads to state `t` and (2) executing `T` in state `t` leads to state `u`, then executing the sequential composition `S; T` in state `s` leads to state `u`.

The conditions (1) and (2) correspond to the two premises of Seq.

The most complicated rule is undoubtedly While-True. Intuitively, it can be understood as follows:

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

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

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

```
inductive big_step : stmt × state → state → Prop
| skip {s} :
  big_step (stmt.skip, s) s
| assign {x a s} :
  big_step (stmt.assign x a, s) (s{x ↦ a s})
| seq {S T s t u} (hS : big_step (S, s) t)
    (hT : big_step (T, t) u) :
  big_step (S ;; T, s) u
| ite_true {b : state → Prop} {S T s t} (hcond : b s)
    (hbody : big_step (S, s) t) :
  big_step (stmt.ite b S T, s) t
| ite_false {b : state → Prop} {S T s t} (hcond : ¬ b s)
    (hbody : big_step (T, s) t) :
  big_step (stmt.ite b S T, s) t
| while_true {b : state → Prop} {S s t u} (hcond : b s)
    (hbody : big_step (S, s) t)
    (hrest : big_step (stmt.while b S, t) u) :
```

```
    big_step (stmt.while b S, s) u
| while_false {b : state → Prop} {S s} (hcond : ¬ b s) :
    big_step (stmt.while b S, s) s
```

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

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 rules that are traditionally used in the scientific literature. If we were to attempt a recursive definition such as

```
def eval : stmt → state → state
| stmt.skip         s := s
| (stmt.assign x a) s := s{x ↦ a s}
| (stmt.ite b S T)  s := if b s then eval S s else eval T s
| (S ;; T)          s := eval T (eval S s)
| (stmt.while b S)  s :=
  if b s then eval (stmt.while b S) (eval S s) else s
```

instead, we would face nontermination of the `stmt.while` case. Indeed, since the program `stmt.while (λ_, true) stmt.skip` loops forever, trying to evaluate it using `eval` would never return. (There are also other issues with the above definition, due to the use of `Prop` to represent Boolean expressions.)

8.4 Properties of the Big-Step Semantics

Equipped with a big-step semantics, we can reason about concrete programs, proving theorems relating final states with initial states. Perhaps more interestingly, we can prove properties of the programming language, such as determinism and nontermination.

We start with determinism:

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

The Lean proof is in the demonstration file associated with this chapter. We content ourselves with an informal proof sketch:

> The proof is by rule induction over `(S, s) ⟹ l`, generalizing over `r`.
>
> Case Skip: The only way to have `(skip, s) ⟹ l` or `(skip, s) ⟹ r` is if `l = r = s`.
>
> Case Asn: Similar to Skip.
>
> Case Seq: We have the hypotheses `(S, s) ⟹ t`, `(T, t) ⟹ l`, `(S, s) ⟹ t'`, and `(T, t') ⟹ r` and the induction hypotheses `∀r, (S, s) ⟹ r → t = r` and `∀r, (T, t) ⟹ r → l = r`. From the first induction hypothesis together with `(S, s) ⟹ t'`, we derive `t = t'`. From the second induction hypothesis together with `(T, t') ⟹ r`, we derive `l = r`.
>
> Case If-True: Since `b s` is true, `(if b then S else T, s) ⟹ r` can only have been derived using If-True and thus `(S, s) ⟹ r`. The induction

hypothesis is `∀r, (S, s) ⟹ r → l = r`. We can apply `(S, s) ⟹ r` to it to obtain `l = r`.

Case If-False: Similar to If-True.

Case While-True: Similar to Seq.

Case While-False: Similar to Skip. □

Given that the WHILE language is deterministic, for the big-step semantics, termination would amount to the following:

```
lemma big_step_terminates {S s} :
  ∃t, (S, s) ⟹ t
```

Literally, this property means that for every statement `S` and state `s`, there exists a state `t` such that executing `S` starting in `s` may terminate in `t`. But because WHILE is deterministic, we can replace "may terminate" by "must terminate."

However, the property does not hold. The big-step semantics can be used to show that `stmt.while (λ_, true) S` will never terminate, regardless of the start state. The proof relies on the `generalize` tactic and on the `generalizing` option to `induction`, both of which were introduced in Section 5.4:

```
lemma big_step_doesnt_terminate {S s t} :
  ¬ (stmt.while (λ_, true) S, s) ⟹ t :=
begin
  generalize hws : (stmt.while (λ_, true) S, s) = ws,
  intro hw,
  induction hw generalizing s,
  case big_step.while_true : b S s t u hcond hbody hrest ih_body
      ih_rest {
    cases hws,
    apply ih_rest,
    refl },
  case big_step.while_false : b S s hcond {
    cases hws,
    apply hcond,
    apply true.intro },
  all_goals { cases hws }
end
```

Notice the use of the "and then" tactic combinator (`;`) to invoke `cases heq` on all subgoals emerging from `induction`. This eliminates all the subgoals except `big_step.while_true` and `big_step.while_false`.

When reasoning about an inductive predicate, it is often convenient to use inversion rules (Section 5.6). Accordingly, we prove the following rules:

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

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

@[simp] lemma big_step_seq_iff {S T s t} :
  (S ;; T, s) ⟹ t ↔ (∃u, (S, s) ⟹ u ∧ (T, u) ⟹ t)
```

```
@[simp] lemma big_step_ite_iff {b S T s t} :
  (stmt.ite b S T, s) ⟹ t ↔
  (b s ∧ (S, s) ⟹ t) ∨ (¬ b s ∧ (T, s) ⟹ t)

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

lemma big_step_while_true_iff {b : state → Prop} {S s u}
    (hcond : b s) :
  (stmt.while b S, s) ⟹ u ↔
  (∃t, (S, s) ⟹ t ∧ (stmt.while b S, t) ⟹ u)

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

We add most of them to the simp set. We leave out `big_step_while_iff` and `big_step_while_true_iff` because they could make `simp` 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 `s` and the final state `t`. This is too coarse-grained to reason about multithreaded programs, where several processes can interact with each other's intermediate states. Moreover, for nondeterministic languages, big-step semantics offer no general way to express termination: A judgment indicates a possibility (executing `S` in state `s` *may* result in state `t`), not a necessity.

Small-step operational semantics give us a finer view of computation. The transition predicate ⇒ has type `stmt × state → stmt × state → Prop`. Intuitively, `(S, s) ⇒ (T, t)` means that executing one step of program `S` in state `s` leaves the program `T` to be executed, in state `t`. If there is no program left to be executed, we put `skip`.

An *execution* is a finite or infinite chain $(S_0, s_0) \Rightarrow (S_1, s_1) \Rightarrow \cdots$ of "small" ⇒ steps. A pair `(S, s)` is called a *configuration*; it is *final* if no transition of the form `(S, s) ⇒ (T, t)` is possible, for any configuration `(T, t)`. 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 valid small-step judgments are specified as derivation rules:

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

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

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

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

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

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

Unlike in the big-step semantics, there is no rule for `skip` in the small-step semantics. This is because a configuration of the form `(skip, s)` is considered final; `skip` is understood as the statement whose execution is vacuous. By inspection of the rules, we can convince ourselves 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 `b` and, depending on its truth value, put the `then` or the `else` branch as the remaining computation to perform.

For the `while` loop, there is a single unconditional rule that expands one iteration of the loop, introducing an `if` statement. It is then the role of the If-True and If-False rules to process the `if`. In the If-True case, we eventually reach the `while` loop again. This can continue forever for infinite loops.

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

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

Based on a small-step semantics, we can *define* a big-step semantics as

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

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

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

8.6 Properties of the Small-Step Semantics

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 and hence that the small-steps semantics cannot get stuck while executing a program. The lemma statement is as follows:

```
lemma small_step_final {S s} :
  (¬ ∃T t, (S, s) ⇒ (T, t)) ↔ S = stmt.skip
```

The proof is by structural induction on `S`.

Like the big-step semantics, the small-step semantics is deterministic:

```
lemma small_step_deterministic {S s Ll Rr}
    (hl : (S, s) ⇒ Ll) (hr : (S, s) ⇒ Rr) :
  Ll = Rr
```

The proof is by rule induction on `hl` or `hr`.

For the small-step semantics, a configuration (S_0, s_0) terminates if all executions starting in it are finite: $(S_0, s_0) \Rightarrow (S_1, s_1) \Rightarrow \cdots \Rightarrow (S_n, s_n)$. It is nonterminating if there exists an infinite chain $(S_0, s_0) \Rightarrow (S_1, s_1) \Rightarrow \cdots$. The programming language as a whole is terminating if and only if all its configurations terminate. It is easy to show that the WHILE language is nonterminating, by taking S_0 `:= stmt.while (λ_, true) stmt.skip`. For any s_0, we then have

$$(S_0, s_0) \Rightarrow (S_0, s_0) \Rightarrow (S_0, s_0) \Rightarrow \cdots$$

We can define inversion rules about the small-step semantics, such as these:

```
lemma small_step_skip {S s t} :
  ¬ ((stmt.skip, s) ⇒ (S, t))

@[simp] lemma small_step_seq_iff {S T s Ut} :
  (S ;; T, s) ⇒ Ut ↔
  (∃S' t, (S, s) ⇒ (S', t) ∧ Ut = (S' ;; T, t))
  ∨ (S = stmt.skip ∧ Ut = (T, s))

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

A more fundamental result is the equivalence between the big-step and the small-step semantics:

```
lemma big_step_iff_star_small_step {S s t} :
  (S, s) ⟹ t ↔ (S, s) ⇒* (stmt.skip, t)
```

Recall that $\Rightarrow*$ denotes the reflexive transitive closure of the small-step predicate $\Rightarrow$. The theorem's proof is beyond the scope of this course. We refer to Chapter 7 of *Concrete Semantics: With Isabelle/HOL* [24] or to the demonstration file accompanying this chapter.

8.7 Parallelism

The WHILE language is a sequential language: one in which statements are executed in turn, without interruptions or interference. In this section, we will consider *PAR*, an extension of WHILE with top-level parallelism. A PAR program consists of a list `[S₁, ..., Sₙ]` of WHILE programs. The individual WHILE programs are executed in parallel, as their own threads, using an interleaving semantics. This means that the semantics of the entire PAR program is given by interleaving small steps of the threads.

A PAR configuration `(Ss, s)` consists of a list `Ss` of WHILE programs and a state `s`. A PAR step can be defined as follows:

```
inductive par_step :
    nat → list stmt × state → list stmt × state → Prop
| intro {Ss Ss' S S' s s' i}
    (hi : i < list.length Ss)
    (hS : S = list.nth_le Ss i hi)
    (hs : (S, s) ⇒ (S', s'))
    (hS' : Ss' = list.update_nth Ss i S') :
  par_step i (Ss, s) (Ss', s')
```

Informally, from an arbitrary configuration `(Ts ++ [S] ++ Us, s)`, we can make a small step to `(Ts ++ [S'] ++ Us, s')` whenever `(S, s) ⇒ (S', s')`. The `par_step` predicate's first argument is the index of the thread that makes a transition.

Clearly, PAR is nondeterministic. Consider the program `[S₀, S₁]`, with

```
S₀ := stmt.assign "x" (λ_, 42)     S₁ := stmt.assign "x" (λ_, 999)
```

and consider the start state `s`. Then taking a `par_step 0` step results in the configuration `([stmt.skip, S₁], s{x ↦ 42})`, whereas taking a `par_step 1` step results in `([S₀, stmt.skip], s{x ↦ 999})`.

Sometimes the nondeterminism is harmless, if the final result is the same. This is called *determinacy*. In particular, some languages enjoy the diamond property:

```
lemma par_step_diamond {i j Ss Ts Ts' s t t'}
    (hi : i < list.length Ss)
    (hj : j < list.length Ss)
    (hij : i ≠ j)
    (hT : par_step i (Ss, s) (Ts, t))
    (hT' : par_step j (Ss, s) (Ts', t')) :
  ∃u Us, par_step j (Ts, t) (Us, u) ∧
    par_step i (Ts', t') (Us, u)
```

The diamond property implies determinacy. However, PAR is not determinate, and hence cannot enjoy the diamond property. If we take `par_step 0` followed by `par_step 1`, we inevitably reach the final state `([stmt.skip, stmt.skip], s{x ↦ 999})`, whereas if we start with `par_step 1` and then take `par_step 0`, we inevitably reach `([stmt.skip, stmt.skip], s{x ↦ 42})`, a counterexample both to determinacy and to the diamond property.

The counterexample has two threads writing to the same variable `x`. If we were to restrict access to variables written by other threads, we could salvage the diamond property. First, we need some auxiliary definitions:

```
def stmt.W : stmt → set string
| stmt.skip         := ∅
| (stmt.assign x _) := {x}
| (stmt.seq S T)    := stmt.W S ∪ stmt.W T
| (stmt.ite _ S T)  := stmt.W S ∪ stmt.W T
| (stmt.while _ S)  := stmt.W S

def exp.R {α : Type} : (state → α) → set string
| f := {x | ∀s n, f (s{x ↦ n}) ≠ f s}

def stmt.R : stmt → set string
| stmt.skip         := ∅
| (stmt.assign _ a) := exp.R a
| (stmt.seq S T)    := stmt.R S ∪ stmt.R T
| (stmt.ite b S T)  := exp.R b ∪ stmt.R S ∪ stmt.R T
| (stmt.while b S)  := exp.R b ∪ stmt.R S

def stmt.V : stmt → set string
| S := stmt.W S ∪ stmt.R S
```

The function `stmt.W` returns the set of variables written by a thread, `stmt.R` is the set of variables read by a thread, and `stmt.V` is the set of variables written or read by a thread. Notice the markedly semantic definition of `exp.R`: A variable `x` is read by a shallowly embedded arithmetic or Boolean expression `f` if there exists a state `s` and a value `n` such that setting `x` to `n` in `s` influences `f`'s behavior. If we had a deep embedding of arithmetic and Boolean expressions, we could perform a syntactic analysis instead (as we do in `stmt.R` for programs).

Using the above definitions, we can revise the diamond property:

```
lemma par_step_diamond_VW_disjoint {i j Ss Ts Ts' s t t'}
    (hiS : i < list.length Ss)
    (hjT : j < list.length Ts)
    (hij : i ≠ j)
    (hT : par_step i (Ss, s) (Ts, t))
    (hT' : par_step j (Ss, s) (Ts', t'))
    (hWV : stmt.W (list.nth_le Ss i hiS)
       ∩ stmt.V (list.nth_le Ts j hjT) = ∅)
    (hVW : stmt.V (list.nth_le Ss i hiS)
       ∩ stmt.W (list.nth_le Ts j hjT) = ∅) :
  ∃u Us, par_step j (Ts, t) (Us, u) ∧
    par_step i (Ts', t') (Us, u)
```

Intuitively, there are reasons to believe that this proposition is valid, but we have not carried out the proof. Considering Conrad Watt's experience with WebAssembly and that of dozens if not hundreds of other researchers, we should not put too much faith in the proposition until we have established it formally.

Chapter 9

Hoare Logic

If operational semantics corresponds to an idealized interpreter, *Hoare logic* corresponds to an idealized verifier. Hoare logic can be used to specify the semantics of a programming language, but it is particularly convenient to reason about concrete programs and prove them correct. It is named after its inventor, Charles Antony Richard (Tony) Hoare. It is also called *axiomatic semantics*.

This chapter is inspired by Chapter 12 of *Concrete Semantics: With Isabelle/HOL* [24].

9.1 Hoare Triples

Hoare logic is a framework for deducing valid correctness formulas in a mechanical way, using a set of derivation rules. It allows us to reason directly about a program's syntax, without concerning ourselves with its operational semantics. The approach is mechanical in the sense that the applicability of a derivation rule can easily be checked.

We start by introducing Hoare logic abstractly, without any connection to Lean. In a second step, we will see how we can embed Hoare logic judgments in Lean in a natural 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 program variables. For the moment, we imagine the formulas as syntactic objects built using the familiar connectives and quantifiers. The intended meaning of a Hoare triple is as follows:

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

This is a *partial correctness* statement: The program is correct *if* it terminates normally; otherwise, we do not know or care. For WHILE programs, the only way not to terminate normally is to enter an infinite loop. For other programming languages, infinite recursion and run-time errors such as division by zero may also result in divergence or abnormal termination.

Intuitively, all of the Hoare triples below should be valid:

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

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

$$\{\mathtt{b} \geq 5\}\ \mathtt{b} := \mathtt{b} + 1\ \{\mathtt{b} \geq 6\}$$

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

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

The first three Hoare triples should be fairly natural. The fourth triple is vacuously true, since the precondition `false` can never be met. The part "If the precondition `P` is true" of the definition of Hoare triple is always false; hence the triple is true. The triple is equivalent to the proposition `false` $\rightarrow$ `b = 10`, which holds for any value of `b`. As for the fifth triple, there are no guarantees that control will escape the loop (if `i > 10` initially), but if it does escape, then the loop's condition must be false and hence we have the postcondition `i = 10`.

Four further interesting triples:

$$\{\texttt{true}\}\,\texttt{S}\,\{\texttt{true}\}$$

$$\{\texttt{true}\}\,\texttt{S}\,\{\texttt{false}\}$$

$$\{\texttt{false}\}\,\texttt{S}\,\{\texttt{true}\}$$

$$\{\texttt{false}\}\,\texttt{S}\,\{\texttt{false}\}$$

The first triple is always true (and therefore pointless), regardless of `S`. The second triple is true only if `S` never terminates (e.g., `S := while true do skip`). The third and fourth triples are true of any statement `S`: The precondition is never satisfied, so any postcondition holds vacuously.

9.2 Hoare Rules

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

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

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

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

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

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

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

In the Asn rule, the syntax `Q[a/x]` denotes the condition `Q` in which all occurrences of `x` are replaced by `a`. Another way to present the rule is as follows:

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

where `[x]` factors out all occurrences of `x` in `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 below:

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

Using elementary arithmetic, we can simplify the computed preconditions; for example, $0 = 0$ is equivalent to true, and $x + 1 \geq 5$ is equivalent to $x \geq 4$.

The SEQ rule requires us to come up with an intermediate condition R that holds after executing S and before executing T. Here is an example of SEQ in action:

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

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

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

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

Reading the tree from top to bottom, we have strengthened the triple's precondition from $x > 4$ to $x > 8$ and weakened the postcondition from $y > 4$ to $y > 0$. We can also read the tree from the bottom up: To prove the triple $\{x > 8\}\ y := x\ \{y > 0\}$, it suffices to prove $\{x > 4\}\ y := x\ \{y > 4\}$, where the precondition is weakened and the postcondition is strengthened.

Except for CONSEQ, the rules are *syntax-driven*: We know which rule to apply in each case, simply by inspecting the statement at hand. For an assignment, we apply ASN; for a `while` loop, we apply WHILE; and so on.

The rules SEQ, IF, and CONSEQ are *bidirectional*: Their conclusions are of the form $\{P\}\ \ldots\ \{Q\}$, for distinct mathematical metavariables P, Q. This can make them convenient to apply. By combining the other rules with CONSEQ, we can derive bidirectional variants:

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

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

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

As an easy exercise, you could try to derive each of these rules from Skip, Asn, or While in conjunction with Conseq.

9.3 A Semantic Approach to Hoare Logic

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

> If the precondition `P` is true before `S` is executed (i.e., `P s`) and the execution terminates normally (i.e., `(S, s) ⟹ t`), the postcondition `Q` is true at termination (i.e., `Q t`).

Instead of pursuing this approach, we propose to define Hoare triples semantically in Lean, in terms of the big-step semantics, so that they are *correct by definition*. Then we will derive the Hoare rules as lemmas, instead of stating them as introduction rules. In conjunction with the use of predicates to represent formulas, this approach is resolutely semantic.

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

```
def partial_hoare (P : state → Prop) (S : stmt)
  (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} :
  {* P *} stmt.skip {* P *}

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

lemma seq_intro {P Q R S T} (hS : {* P *} S {* Q *})
    (hT : {* Q *} T {* R *}) :
  {* P *} S ;; T {* R *}

lemma ite_intro {b P Q : state → Prop} {S T}
    (hS : {* λs, P s ∧ b s *} S {* Q *})
    (hT : {* λs, P s ∧ ¬ b s *} T {* Q *}) :
  {* P *} stmt.ite b S T {* Q *}
```

```
lemma while_intro (P : state → Prop) {b : state → Prop} {S}
    (h : {* λs, P s ∧ b s *} S {* P *}) :
  {* P *} stmt.while b S {* λs, P s ∧ ¬ b s *}

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

All of the above lemmas have proofs based on the big-step semantics. Some of the triples—for example, the precondition in `assign_intro`—need to refer to the state explicitly. In such case, we use a λ-expression to access it. Recall that `P` and `λs, P s` are the same by η-conversion. Moreover, for a 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. As in Chapter 8, the syntax `s{x ↦ n}` in the assignment rule denotes the state that is identical to `s` at all points except `x` and that maps `x` to `n`.

The following convenience rules can be derived from the core rules:

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

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

lemma skip_intro' {P Q : state → Prop} (h : ∀s, P s → Q s) :
  {* P *} stmt.skip {* Q *}

lemma assign_intro' {P Q : state → Prop} {x} {a : state → ℕ}
    (h : ∀s, P s → Q (s{x ↦ a s})):
  {* P *} stmt.assign x a {* Q *}

lemma seq_intro' {P Q R S T} (hT : {* Q *} T {* R *})
    (hS : {* P *} S {* Q *}) :
  {* P *} S ;; T {* R *}

lemma while_intro' {b P Q : state → Prop} {S}
    (I : state → Prop)
    (hS : {* λs, I s ∧ b s *} S {* I *})
    (hP : ∀s, P s → I s)
    (hQ : ∀s, ¬ b s → I s → Q s) :
  {* P *} stmt.while b S {* Q *}
```

Using the bidirectional `assign_intro'`, we can derive a forward version of the assignment rule:

```
lemma assign_intro_forward (P) {x a} :
  {* P *}
  stmt.assign x a
  {* λs, ∃n₀, P (s{x ↦ n₀}) ∧ s x = a (s{x ↦ n₀}) *} :=
```

```
begin
  apply assign_intro',
  intros s hP,
  apply exists.intro (s x),
  simp [*]
end
```

The bound variable n_0 represents the value of x before the assignment. Therefore, in the postcondition, $s\{x \mapsto n_0\}$ is the state before the assignment. Since P is a precondition, we have $P\ (s\{x \mapsto n_0\})$. In addition, the new value of x, given by s x, must be equal to the value of the arithmetic expression a evaluated in the old state $s\{x \mapsto n_0\}$.

The forward rule is less convenient than the backward rule, because the postcondition contains an existential quantifier. It is possible to state a backward rule in a similar style, revealing a hidden symmetry:

```
lemma assign_intro_backward (Q : state → Prop) {x}
    {a : state → ℕ} :
  {* λs, ∃n', Q (s{x ↦ n'}) ∧ n' = a s *}
  stmt.assign x a
  {* Q *}
```

Notice that this existential quantifier can be eliminated using a one-point rule (Section 3.3). The result is the familiar backward rule `assign_intro`.

9.4 First Program: Exchanging Two Variables

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

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

The correctness statement is as follows:

```
lemma SWAP_correct (a₀ b₀ : ℕ) :
  {* λs, s "a" = a₀ ∧ s "b" = b₀ *}
  SWAP
  {* λs, s "a" = b₀ ∧ s "b" = a₀ *}
```

The logical variables a_0 and b_0 "freeze" the initial value of the program variables a and b so that we can refer to them in the postcondition. It would make no sense to put `λs, s "a" = s "b" ∧ s "b" = s "a"` as the postcondition.

The correctness proof follows:

```
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 the sequential composition and assignment rules are guided by the program's syntax. There are two sequential compositions and three assignments in the program and therefore as many invocations of the corresponding rules. We end up with a very ugly subgoal:

```
⊢ ∀s : state,
    s "a" = a₀ ∧ s "b" = b₀ →
    s{"t" ↦ s "a"}{"a" ↦ s{"t" ↦ s "a"} "b"}
      {"b" ↦ s{"t" ↦ s "a"}{"a" ↦ s{"t" ↦ s "a"} "b"} "t"} "a" = b₀ ∧
    s{"t" ↦ s "a"}{"a" ↦ s{"t" ↦ s "a"} "b"}
      {"b" ↦ s{"t" ↦ s "a"}{"a" ↦ s{"t" ↦ s "a"} "b"} "t"} "b" = a₀
```

Fortunately, `simp` can simplify the subgoal dramatically, and if we enable *contextual rewriting*—an option that tells `simp` to use any assumptions in the goal's target (such as `s "a" = a₀` and `s "b" = b₀`) as left-to-right rewrite rules—`simp` can even prove the subgoal.

The above proof could also be carried out directly in terms of the big-step semantics, without using the Hoare rules:

```
lemma SWAP_correct₂ (a₀ b₀ : ℕ) :
  {* λs, s "a" = a₀ ∧ s "b" = b₀ *}
  SWAP
  {* λs, s "a" = b₀ ∧ s "b" = a₀ *} :=
begin
  intros s t hP hstep,
  cases hstep,
  cases hstep_hS,
  cases hstep_hT,
  cases hstep_hT_hS,
  cases hstep_hT_hT,
  finish
end
```

The `finish` tactic applies a combination of techniques, including normalization of logical connectives and quantifiers, simplification, congruence closure, and quantifier instantiation. Like `tautology` (Section 6.8), it provides strong automation that can ruin many logical exercises if introduced too early in the course.

9.5 Second Program: 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 : stmt :=
stmt.while (λs, s "n" ≠ 0)
  (stmt.assign "n" (λs, s "n" - 1) ;;
   stmt.assign "m" (λs, s "m" + 1))
```

Because of the presence of a `while` loop, 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 *} :=
partial_hoare.while_intro' (λs, s "n" + s "m" = n₀ + m₀)
  begin
    apply partial_hoare.seq_intro',
    { apply partial_hoare.assign_intro },
    { apply partial_hoare.assign_intro',
      simp,
      intros s hnm hnz,
      rewrite ←hnm,
      cases s "n"; finish }
  end
  (by simp { contextual := true })
  (by simp { contextual := true })
```

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 `n` and `m`, as required by the precondition.

How did we come up with this invariant? Even for a simple loop, finding a suitable invariant can be challenging. The difficulty is that the invariant must be

1. provably true before we enter the loop;
2. remain true after each iteration of the loop if it was true before the iteration;
3. be strong enough to imply the desired postcondition of the loop.

An invariant such as `true` meets requirements 1 and 2 but usually not 3. Similarly, `false` meets requirements 2 and 3 but usually not 1. In practice, suitable invariants tend to be of the form

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

where + in general denotes some appropriate operator and not necessarily addition. When we enter the loop, "work already done" will often be 0 (or some other appropriate "zero" value), and the invariant becomes

work remaining to be done = desired result

This invariant would have to be provable at the beginning of the loop—either from the postcondition of the previous statement or from the desired precondition of the entire program if there is no previous statement. When we exit the loop, "work remaining to be done" should be 0 (or some variant thereof), and the invariant becomes

work already done = desired result

Often, "work already done" takes the form of a variable in which we accumulate the result, whereas "work remaining to be done" is similar to "desired result" but depends on program variables whose values change inside the loop and must account for "work already done."

For the ADD program's loop, the "work already done" is m, the "work remaining to be done" is n, and the "desired result" is $n_0 + m_0$. When entering the loop, the invariant $m + n = n_0 + m_0$ holds because then $m = m_0$ and $n = n_0$. (Observe that the "work already done" is not 0 for this example, because we reuse the input m as our accumulator, as an optimization.) When exiting the loop, we have that $n = 0$, so the invariant becomes $m = n_0 + m_0$. We can retrieve the desired result from m.

The application of `while_intro'` is performed directly as a proof term. It gives rise to three subgoals. The proofs that the invariant is implied by the desired precondition and that it implies the desired postcondition are trivial: They consist of a call to `simp` with contextual rewriting enabled. The only nontrivial subgoal is the condition 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.

For this example, Hoare logic really helps. Reasoning directly about the operational semantics would be a less appealing prospect, because we would need induction to reason about the `while` loop. With Hoare logic, this induction is carried out once and for all in the proof of the `while_intro` rule.

9.6 Finish Tactic

finish

`finish` [`[`h_1`, ..., `h_m`]`] [`using [`k_1`, ..., `k_n`]`]

The `finish` tactic first normalizes and simplifies the hypotheses, using the optionally supplied lemmas h_1, ..., h_m. Then it applies the congruence closure (`cc`) in combination with the optionally supplied lemmas k_1, ..., k_n. Universally quantified hypotheses and lemmas are instantiated using heuristics. Like `cc`, `finish` either fully succeeds or fails. The tactic is complete for propositional logic.

9.7 A Verification Condition Generator

Verification condition generators (VCGs) are programs that apply Hoare logic rules, producing *verification conditions* that must be proved manually. Users 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]

VCGs typically work backwards from the postcondition, using *backward* rules—rules stated to have an arbitrary Q as their postcondition. This works well because the central rule of Hoare logic—the assignment rule—is backward. (If it were not for the assignment statement, the state would never change, and there would be no need for Hoare logic.)

We can use Lean's metaprogramming framework to define a simple VCG. First, we introduce a constant called `stmt.while_inv` that carries a user-supplied invariant I in addition to the loop condition b and body S:

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

```
def stmt.while_inv (I b : state → Prop) (S : stmt) : stmt :=
stmt.while b S
```

We provide two Hoare rules for the construct: a backward rule and a bidirectional rule. Both are justified in terms of the bidirectional `while_intro'` rule:

```
lemma while_inv_intro {b I Q : state → Prop} {S}
    (hS : {* λs, I s ∧ b s *} S {* I *})
    (hQ : ∀s, ¬ b s → I s → Q s) :
  {* I *} stmt.while_inv I b S {* Q *} :=
while_intro' I hS (by cc) hQ

lemma while_inv_intro' {b I P Q : state → Prop} {S}
    (hS : {* λs, I s ∧ b s *} S {* I *})
    (hP : ∀s, P s → I s) (hQ : ∀s, ¬ b s → I s → Q s) :
  {* P *} stmt.while_inv I b S {* Q *} :=
while_intro' I hS hP hQ
```

The rules simply use the invariant annotation as their invariant. When using the framework, we will have to be careful to annotate all `while` loops with suitable invariants, because there will not be a second chance to provide them.

The code of the VCG is fairly concise:

```
meta def vcg : tactic unit :=
do
  t ← tactic.target,
  match t with
  | `({* %%P *} %%S {* _ *}) :=
    match S with
    | `(stmt.skip)           :=
      tactic.applyc
        (if expr.is_mvar P then ``partial_hoare.skip_intro
         else ``partial_hoare.skip_intro')
    | `(stmt.assign _ _)     :=
      tactic.applyc
        (if expr.is_mvar P then ``partial_hoare.assign_intro
         else ``partial_hoare.assign_intro')
    | `(stmt.seq _ _)        :=
      tactic.applyc ``partial_hoare.seq_intro'; vcg
    | `(stmt.ite _ _ _)      :=
      tactic.applyc ``partial_hoare.ite_intro; vcg
    | `(stmt.while_inv _ _ _) :=
      tactic.applyc
        (if expr.is_mvar P then ``partial_hoare.while_inv_intro
         else ``partial_hoare.while_inv_intro');
        vcg
    | _                      :=
      tactic.fail (to_fmt "cannot analyze " ++ to_fmt S)
    end
  | _                       := pure ()
  end
```

The VCG proper extracts the current goal's target and inspects it. If it is a Hoare triple, it inspects its precondition `P` and statement `S`. If the precondition is a metavariable (e.g., `?P`), it applies a backward rule if there exists one (e.g., the standard assignment rule `assign_intro`) because this will instantiate the metavariable. Otherwise, a bidirectional rule is used, with an arbitrary variable as its precondition, which can be matched against the target's precondition. For `while` loops, we only consider programs that use `stmt.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.

9.8 Second Program Revisited: Adding Two Numbers

Using the verification condition generator, we can revisit the correctness proof for the `ADD` program 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 *}
     stmt.while_inv (λs, s "n" + s "m" = n₀ + m₀)
       (λs, s "n" ≠ 0)
       (stmt.assign "n" (λs, s "n" - 1) ;;
        stmt.assign "m" (λs, s "m" + 1))
     {* λs, s "n" = 0 ∧ s "m" = n₀ + m₀ *}, from
  begin
    vcg; simp { contextual := tt },
    intros s hnm hnz,
    rewrite ←hnm,
    cases s "n"; finish
  end
```

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 `stmt.while` by `stmt.while_inv`, which is equal to `stmt.while` by definition. The program and its pre- and postconditions are otherwise the same as in the lemma statement.

We invoke `vcg` as the first proof step. This will apply all the necessary Hoare rules and leave us with some subgoals. We first simplify them with contextual rewriting enabled; this already takes care of two of the three subgoals. For the third subgoal, the proof is as in Section 9.5.

9.9 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 you are asked to provide a sorting algorithm at an exam, you should certainly not give `while true do skip` as the answer.

Total correctness is a stronger notion that asserts, in addition to partial correctness, 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, this can be formulated equivalently as follows:

> If the precondition `P` holds before `S` is executed, there exists a state in which execution terminates normally and the postcondition `Q` holds in that state.

Here is an example Hoare triple that is valid:

$$\texttt{[i} \leq \texttt{10] while i < 10 do i := i + 1 [i = 10]}$$

For the WHILE language, the distinction between partial and total correctness only concerns `while` loops (and programs containing them). The Hoare rule for `while` must now be annotated by a *variant* `V`—a natural number that decreases with each iteration:

$$\frac{\texttt{[I} \wedge \texttt{b} \wedge \texttt{V = } v_0\texttt{] S [I} \wedge \texttt{V < } v_0\texttt{]}}{\texttt{[I] while b do S [I} \wedge \neg\texttt{b]}}\,\textsc{While}$$

The variable v_0 is a logical variable that freezes `V`'s initial value and whose scope is the entire premise, whereas `V` is a mathematical metavariable (like `I`, `b`, and `S`). For the example above, we would take `10 - i` as the variant. A derivation tree for this example is given on Wikipedia.[2]

9.10 Summary of New Constructs

Tactics

`finish` normalizes, simplifies, performs congruence closure, and more

`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 attempt to directly specify the meaning of programs as a mathematical object. If operational semantics corresponds to an idealized interpreter and Hoare logic corresponds to an idealized verifier, then denotational semantics corresponds to an idealized compiler.

The core of this chapter is modeled closely after Chapter 11 of *Concrete Semantics: With Isabelle/HOL* [24].

10.1 Compositionality

A *denotational semantics* defines the meaning of each program as a mathematical object. Abstractly, it can be regarded as a function

$$[\![\ ]\!] : \text{syntax} \to \text{semantics}$$

A key property of denotational semantics is *compositionality*: The meaning of a compound statement should be defined in terms of the *meaning* of its components. Consider the trivial definition

$$[\![\texttt{S}]\!] = \{(\texttt{s}, \texttt{t}) \mid (\texttt{S}, \texttt{s}) \Longrightarrow \texttt{t}\}$$

in terms of the big-step semantics ($\Longrightarrow$). Actually, because Lean supports only variables on the left-hand side of the vertical bar, we must write

$$[\![\texttt{S}]\!] = \{\texttt{st} \mid (\texttt{S}, \texttt{prod.fst st}) \Longrightarrow \texttt{prod.snd st}\}$$

This definition specifies the desired semantics, but it does not qualify as a denotational semantics due to lack of compositionality: The meaning of the compound statements (sequential composition, `if then else`, and `while`) is given directly, without using the denotation of their components.

A fully compositional definition makes it possible to reason equationally about programs, which is often more convenient than using the introduction, elimination, and inversion principles of $\Longrightarrow$. In essence, we want structurally recursive equations of the form

$$\begin{aligned} [\![\texttt{S ; T}]\!] &= \ldots [\![\texttt{S}]\!] \ldots [\![\texttt{T}]\!] \ldots \\ [\![\texttt{if b then S else T}]\!] &= \ldots [\![\texttt{S}]\!] \ldots [\![\texttt{T}]\!] \ldots \\ [\![\texttt{while b do S}]\!] &= \ldots [\![\texttt{S}]\!] \ldots \end{aligned}$$

with no occurrences of S and T in the right-hand sides except as arguments to ⟦ ⟧.

An evaluation function such as

$$\texttt{eval} : \underbrace{\texttt{aexp}}_{\text{syntax}} \to \underbrace{(\texttt{string} \to \mathbb{Z}) \to \mathbb{Z}}_{\text{semantics}}$$

on arithmetic expressions is a denotational semantics. Recall that it satisfies recursive equations such as

$$\texttt{eval}\ (\texttt{aexp.add}\ e_1\ e_2) = \texttt{eval}\ e_1 + \texttt{eval}\ e_2$$

Such equations are compositional: The semantics of $\texttt{aexp.add}\ e_1\ e_2$ is defined in terms of the semantics of e_1 and e_2. Incidentally, we could have defined an operational semantics for arithmetic expressions, but it would have been more cumbersome to formulate and reason about.

Denotational semantics are a natural match for arithmetic expressions but also functional programs. Now we want a convenient denotational semantics for imperative programs. Because of `while` loops, which are not guaranteed to terminate, we need to develop some additional mathematical notions to reach the point where we can formulate the desired semantics.

10.2 A Relational Denotational Semantics

Denotational semantics for deterministic languages are normally given as a function from prestate to poststate, but relations are more general and more convenient to manipulate. We present a relational denotational semantics.

A denotational semantics of a program will be a mathematical object of type $\texttt{set}\ (\texttt{state} \times \texttt{state})$. The relational approach was also used for the big-step semantics, which took the form of a predicate of type $\texttt{state} \to \texttt{state} \to \texttt{Prop}$.These two types are isomorphic, but $\texttt{set}\ \alpha$, defined as $\alpha \to \texttt{Prop}$, supports convenient operations and notations, such as $\emptyset$, $\cup$, $\cap$, $\in$, and $\{\ldots \mid \ldots\}$ (Section 6.7).

Our semantics is called `denote`, with ⟦ ⟧ as syntactic sugar. We start with the first four equations of the definition, keeping `while` for later:

```
def denote : stmt → set (state × state)
| stmt.skip         := Id
| (stmt.assign x a) :=
  {st | prod.snd st = (prod.fst st){x ↦ a (prod.fst st)}}
| (stmt.seq S T)    := denote S ○ denote T
| (stmt.ite b S T)  := (denote S ⨾ b) ∪ (denote T ⨾ (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.

The semantics of sequential composition is elegantly expressed as a relational composition ○, which is defined such that

$$r_1 \bigcirc r_2 = \{ac \mid \exists b, (\texttt{prod.fst}\ ac, b) \in r_1 \wedge (b, \texttt{prod.snd}\ ac) \in r_2\}$$

The semantics of `if then else` is the union of the semantics of the `then` and the `else` branch, restricted to include only the tuples whose first component makes the Boolean condition true or false, depending on the branch. The restriction operator is defined by

```
r ⌋ p = {ab | p (prod.fst ab) ∧ ab ∈ r}
```

The difficulties arise when we try to define the semantics of `while` loops. We would like to write

```
| (stmt.while b S)  :=
  ((denote S ○ denote (stmt.while b S)) ⌋ b)
  ∪ (Id ⌋ (λs, ¬ b s))
```

but this is not well founded due to the recursive call on `stmt.while b S`. We need to write this differently. What we are looking for on the right-hand side is some term `X` that satisfies the equation

```
X = ((denote S ○ X) ⌋ b) ∪ (Id ⌋ (λs, ¬ b s))
```

In mathematical jargon, we are looking for a fixpoint. The next four sections are concerned with building an operator `lfp` that computes the fixpoint for a given fixpoint equation. Using `lfp`, we will be able to define the semantics of `while` loops as follows:

```
| (stmt.while b S)  :=
  lfp (λX, ((denote S ○ X) ⌋ b) ∪ (Id ⌋ (λs, ¬ b s)))
```

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 several 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, with `X : ℕ → Prop`:

```
X = (λn : ℕ, n = 0 ∨ (∃m : ℕ, n = m + 2 ∧ X m))
```

This equation is the β-contracted version of an equation that obviously has the right format:

```
X = (λ(p : ℕ → Prop) (n : ℕ), n = 0 ∨ (∃m : ℕ, n = m + 2 ∧ p m)) X
                                  ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ f
```

A solution is `X := even`, the predicate that characterizes the even natural numbers. Recall that in Section 5.6, we proved the inversion rule

```
even n ↔ n = 0 ∨ (∃m : ℕ, n = m + 2 ∧ even m)
```

For this example, it turns out that `even` is the only fixpoint. In general, the least and greatest fixpoint may be different. Consider the equation

$$X = (\lambda p,\ p)\ X$$

for $X : \mathbb{N} \to \mathtt{Prop}$. The least fixpoint is $\lambda \_,\ \mathtt{False}$ and the greatest fixpoint is $\lambda \_,\ \mathtt{True}$. By convention, we have $\mathtt{False} < \mathtt{True}$ and thus $(\lambda \_,\ \mathtt{False}) < (\lambda \_,\ \mathtt{True})$. Similarly, $\emptyset <$ `@set.univ` α for any inhabited type α.

For the semantics of `while` loops:

- X will have type `set (state` $\times$ `state)` of 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 executed, it must terminate after some number of iterations `n`. Let f^n denote the `n`-fold composition of `f`, corresponding to performing at most `n` iterations. We have $X = f^n(\emptyset)$ in this case. If we do not know the value of `n` ahead of time, we can always take

$$X = f^0(\emptyset) \cup f^1(\emptyset) \cup f^2(\emptyset) \cup \cdots$$

Kleene's fixpoint theorem[1] states that such an X is equal to the least fixpoint `lfp f`. The greatest fixpoint would also consider cyclic and diverging executions, which we do not want.

Incidentally, Lean's inductive predicates correspond to least fixpoints, but they are built into Lean's logic (the calculus of inductive constructions), without using a fixpoint operator like `lfp`.

10.4 Monotone Functions

We claimed above that the least and greatest fixpoints are guaranteed to exist under some conditions on `f`. It is time to make this more precise. Let α and β be arbitrary types, each equipped with a partial order $\le$. The function $f : \alpha \to \beta$ is *monotone* if $a \le b \to f\ a \le f\ b$ for all `a`, `b`. The function $f : \mathtt{set}\ \alpha \to \mathtt{set}\ \alpha$ admits least and greatest fixpoints if `f` is monotone.

Many operations on sets (e.g., union $\cup$), relations (e.g., composition $\circ$), and functions (e.g., the identity function $\lambda x,\ x$, the constant function $\lambda \_,\ k$, composition $\circ$) are monotone or preserve monotonicity. Of course, not all functions are monotone. Here is an example of a nonmonotone function `f` on `set` α:

$$f\ A = (\mathtt{if}\ A = \emptyset\ \mathtt{then}\ \mathtt{set.univ}\ \mathtt{else}\ \emptyset)$$

If α is inhabited, we have $\emptyset \subseteq$ `set.univ` but $f\ \emptyset = \mathtt{set.univ} \not\subseteq \emptyset = f\ \mathtt{set.univ}$.

10.5 Complete Lattices

To define `lfp` for sets (including relations), we need two operations: subset $\subseteq$: `set` $\alpha \to$ `set` $\alpha \to$ `Prop` and big intersection $\bigcap$: `set (set` α`)` $\to$ `set` α, which can be defined as follows:

$$\bigcap \mathcal{A} = \{a \mid \forall A,\ A \in \mathcal{A} \to a \in A\}$$

[1] https://en.wikipedia.org/wiki/Kleene_fixed-point_theorem

If $\mathcal{A}$ is a finite set $\{A_1, \ldots, A_n\}$, then $\bigcap \mathcal{A} = A_1 \cap \cdots \cap A_n$.

We can define `lfp` more generally so that it works not only with sets but with any instance of the algebraic structure called a complete lattice. A *complete lattice* α is an ordered type for which each `set` α has an infimum, also called *greatest lower bound*. A complete lattice consists of

1. a partial order $\leq : \alpha \to \alpha \to$ `Prop` (i.e., a reflexive, transitive, and antisymmetric binary predicate);
2. an operator $\bigsqcap$: `set` $\alpha \to \alpha$, called the *infimum*.

The $\bigsqcap$ operator satisfies the following two conditions:

1. $\bigsqcap$ `A` is a lower bound of `A`: $\bigsqcap$ `A` $\leq$ `b` for all `b` $\in$ `s`;
2. $\bigsqcap$ `A` is a greatest lower bound: `b` $\leq \bigsqcap$ `A` for all `b` such that $\forall$`a, a` $\in$ `A` $\to$ `b` $\leq$ `a`.

Together, conditions 1 and 2 ensure that $\bigsqcap$ `A` is the unique greatest lower bound.

The lattice operators $\leq$ and $\bigsqcap$ generalize $\subseteq$: `set` $\alpha \to$ `set` $\alpha \to$ `Prop` and $\bigcap$: `set (set` α`)` $\to$ `set` α. Be aware that $\bigsqcap$ `A` is not necessarily an element of `A`. For example, an open interval $]a, b[$ over $\mathbb{R}$ has infimum $a \notin]a, b[$.

Here are some examples of complete lattices:

- `set` α with respect to $\subseteq$ and $\bigcap$ for all types α;
- `Prop` with respect to $\to$ and λ`A, `$\forall$`a` $\in$ `A, a`;
- `enat` := $\mathbb{N} \cup \{\infty\}$ with respect to $\leq$ and a suitable infimum operator;
- `ereal` := $\mathbb{R} \cup \{-\infty, \infty\}$ with respect to $\leq$ and a suitable infimum operator.

If α is a complete lattice, then $\beta \to \alpha$ is also a complete lattice. If α and β are complete lattices, then $\alpha \times \beta$ is also a complete lattice. In both cases, $\leq$ and $\bigsqcap$ are defined componentwise.

Here are some *non*examples of complete lattices: $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$, and $\mathbb{R}$ with respect to $\leq$. The issue is that there is no greatest element, which $\bigsqcap \emptyset$ must be. Another nonexample: `erat` := $\mathbb{Q} \cup \{-\infty, \infty\}$. The issue is that $\bigsqcap$ `{q | 2 < q * q} = sqrt 2` is not in `erat`.

In Lean, it is natural to use type classes to model complete lattices:

```
@[class] structure complete_lattice (α : Type)
  extends partial_order α : Type :=
(Inf    : set α → α)
(Inf_le : ∀A b, b ∈ A → Inf A ≤ b)
(le_Inf : ∀A b, (∀a, a ∈ A → b ≤ a) → b ≤ Inf A)
```

The type `set` α is an instance of the type class:

```
@[instance] def set.complete_lattice {α : Type} :
  complete_lattice (set α) :=
{ le          := (⊆),
  le_refl     := by tautology,
  le_trans    := by tautology,
  le_antisymm :=
    begin
      intros A B hAB hBA,
      apply set.ext,
      tautology
    end,
```

```
Inf         := λ𝒜, {a | ∀A, A ∈ 𝒜 → a ∈ A},
Inf_le      := by tautology,
le_Inf      := by tautology }
```

10.6 Least Fixpoint

With complete lattices in place, we can define the least fixpoint operator:

$$\texttt{lfp f} = \bigsqcap \{\texttt{x} \mid \texttt{f x} \leq \texttt{x}\}$$

The Knaster–Tarski theorem,[2] which we briefly mentioned in Section 5.1, gives us the following properties for any monotone function `f` on a complete lattice:

- `lfp f` is a fixpoint: `lfp f = f (lfp f)`;
- `lfp f` is smaller than any other fixpoint: `X = f X → lfp f ≤ X`.

10.7 A Relational Denotational Semantics, Continued

With `lfp`, we can fulfill our promise and complete the definition of the denotational semantics of WHILE programs:

```
| (stmt.while b S)  :=
  lfp (λX, ((denote S ○ X) ⨽ b) ∪ (Id ⨽ (λs, ¬ b s)))
```

To validate our definition, we can prove the following connection between the denotational and the big-step semantics:

```
lemma denote_iff_big_step (S : stmt) (s t : state) :
  (s, t) ∈ ⟦S⟧ ↔ (S, s) ⟹ t
```

We refer to Chapter 11 of *Concrete Semantics: With Isabelle/HOL* [24] or to the demonstration file accompanying this chapter for the proof.

10.8 Application to Program Equivalence

Based on the denotational semantics, we introduce a notion of program equivalence. Two programs are equivalent if they have the same semantics:

```
def denote_equiv (S₁ S₂ : stmt) : Prop :=
⟦S₁⟧ = ⟦S₂⟧
```

We write `S₁ ~ S₂` as an abbreviation for `denote_equiv S₁ S₂`. It is obvious from the definition that `~` is an equivalence relation.

Program equivalence can be used to replace a subprogram in a larger program with another subprogram as long as they have the same semantics. This is achieved by the following congruence rules:

```
lemma denote_equiv.seq_congr {S₁ S₂ T₁ T₂ : stmt}
    (hS : S₁ ~ S₂) (hT : T₁ ~ T₂) :
  S₁ ;; T₁ ~ S₂ ;; T₂ :=
by simp [denote_equiv, denote, *] at *
```

[2] https://en.wikipedia.org/wiki/Knaster-Tarski_theorem

```
lemma denote_equiv.ite_congr {b} {S₁ S₂ T₁ T₂ : stmt}
    (hS : S₁ ~ S₂) (hT : T₁ ~ T₂) :
  stmt.ite b S₁ T₁ ~ stmt.ite b S₂ T₂ :=
by simp [denote_equiv, denote, *] at *

lemma denote_equiv.while_congr {b} {S₁ S₂ : stmt}
    (hS : S₁ ~ S₂) :
  stmt.while b S₁ ~ stmt.while b S₂ :=
by simp [denote_equiv, denote, *] at *
```

A *congruence rule* is a lemma that lifts an equivalence relation over some context (here, `~` over `stmt.seq`, `stmt.ite`, and `stmt.while`).

Notice how the denotational semantics leads to short proofs by rewriting. This should not be surprising, given that it is designed to be equational and compositional. If we had used the big-step semantics as the basis for program equivalence instead, these proofs would have been much more complicated (as you already surely know if you have done the exercise for Chapter 8).

Equational reasoning is sufficient to prove simple programs equivalent:

```
lemma denote_equiv.skip_assign_id {x} :
  stmt.assign x (λs, s x) ~ stmt.skip :=
by simp [denote_equiv, denote, Id]

lemma denote_equiv.seq_skip_left {S : stmt} :
  stmt.skip ;; S ~ S :=
by simp [denote_equiv, denote, Id, comp]

lemma denote_equiv.seq_skip_right {S : stmt} :
  S ;; stmt.skip ~ S :=
by simp [denote_equiv, denote, Id, comp]
```

We defined the semantics of `while` using the `lfp` operator, but who knows whether monotonicity—which guarantees the existence of a least fixpoint—is met? To quell such doubts, we prove the following lemma:

```
lemma denote_equiv.ite_seq_while {b} {S : stmt} :
  stmt.ite b (S ;; stmt.while b S) stmt.skip ~ stmt.while b S :=
begin
  simp [denote_equiv, denote],
  apply eq.symm,
  apply lfp_eq,
  apply monotone_union,
  { apply sorry_lemmas.monotone_restrict,
    apply sorry_lemmas.monotone_comp,
    { exact monotone_const _ },
    { exact monotone_id } },
  { apply sorry_lemmas.monotone_restrict,
    exact monotone_const _ }
end
```

The lemma gives us a convenient way to expand or contract one iteration of a loop. The second `apply` invokes the lemma `lfp_eq : lfp f = f (lfp f)`, stating that `lfp` is a fixpoint. The rest of the proof is somewhat monotonous: It is all about convincing Lean that `lfp`'s argument is monotone.

Part IV

Mathematics

Chapter 11

Logical Foundations of Mathematics

In this chapter, we dive deeper into the logical foundations of Lean. Most of the features described here are especially relevant for defining mathematical objects and proving theorems about them.

11.1 Universes

In dependent type theory, not only all terms have a type, but also all types have types themselves. We have already seen some occurrences of this principle. The Curry–Howard correspondence tells us to view proofs as terms and propositions as types. For example, the lemma

```
@and.intro : ∀a b, a → b → a ∧ b
```

is really a term `@and.intro` of type `∀a b, a → b → a ∧ b`, which in turn has a type:

```
∀a b, a → b → a ∧ b : Prop
```

What is the type of `Prop` then? `Prop` has the same type as virtually all other types we have constructed so far:

```
Prop : Type
```

What is the type of `Type`? Tthe simplest solution would be to let `Type : Type`, but this choice leads to Girard's paradox, the type theory equivalent of Russel's paradox. To avoid inconsistencies, we need a fresh type to contain `Type`, which we call `Type 1`. `Type 1` itself has type `Type 2`, and so on:

```
Type   : Type 1
Type 1 : Type 2
Type 2 : Type 3
       ⋮
```

In fact, `Type` with no argument is an abbreviation for `Type 0`. If we want to incorporate `Prop` in this hierarchy, we can use the syntax `Sort u`, where `Sort 0` is

an alias for `Prop` and `Sort (u + 1)` is an alias for `Type u`. The hierarchy is captured by the following typing judgment:

$$\frac{}{\texttt{C} \vdash \texttt{Sort u} : \texttt{Sort (u + 1)}}\ \textsc{Sort}$$

All of these types containing other types are called *universes*, and the `u` in the expression `Sort u` is a *universe level*. Although universe levels look like terms of type $\mathbb{N}$, they are in fact neither of type $\mathbb{N}$ nor even terms. Conventionally, universes are capitalized—hence we have `Prop` but `bool`.

Instead of using `Type` everywhere, you can make your theorems slightly more general without having to think about universe levels by writing `Type _` or `Type*`. It leaves it to Lean to figure the right universe level or to create a fresh universe variable. This helps maintain the illusion that we work in a convenient logic where `Type : Type` holds without introducing any paradoxes. In practice, `Type 0` is large enough to formalize most of computer science and mathematics.

11.2 The Peculiarities of Prop

Although `Prop` seems to fit nicely into the hierarchy of universes, it is different from the other universes in several important respects.

11.2.1 Impredicativity

When constructing a new type from other types (e.g., $\alpha \to \beta$ from α `: Type u` and β `: Type v`), the newly constructed type is more complex than each of its components, and it is natural to put it into the largest universe involved (e.g., $\alpha \to \beta$ `: Type (max u v)`). This is exactly what Lean does. The following typing rule expresses this idea generally for dependent types:

$$\frac{\texttt{C} \vdash \alpha : \texttt{Type u} \qquad \texttt{C, a} : \alpha \vdash \beta\ \texttt{a} : \texttt{Type v}}{\texttt{C} \vdash (\forall \texttt{a} : \alpha, \beta\ \texttt{a}) : \texttt{Type (max u v)}}\ \textsc{Forall-Type}$$

This behavior of the `Type` universes is called *predicativity*.

However, it is convenient to have `Prop` behave differently. We would like the expression $\forall$`a : Prop, a` $\to$ `a` to be of type `Prop`—it is, after all, a proposition. Unfolding the syntactic sugar $\to$, this expression is the same as $\forall$`(a : Prop) (x : a), a`. If we had `Type u` instead of `Prop`, the above typing rule would yield

$$(\forall \texttt{(a : Type u) (x : a), a}) : \texttt{Type (u + 1)}$$

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` to be of type `Prop` anyway, we need a special typing rule for $\forall$-expressions with a `Prop` body:

$$\frac{\texttt{C} \vdash \alpha : \texttt{Sort u} \qquad \texttt{C, (a} : \alpha) \vdash \beta\ \texttt{a} : \texttt{Prop}}{\texttt{C} \vdash (\forall \texttt{a} : \alpha, \beta\ \texttt{a}) : \texttt{Prop}}\ \textsc{Forall-Prop}$$

This behavior is called *predicativity* of `Prop`. The rule yields

$$(\forall \texttt{(a : Prop) (x : a), a}) : \texttt{Prop}$$

as desired. The two typing rules for ∀-expressions above can be summarized as the single rule

$$\frac{\texttt{C} \vdash \alpha : \texttt{Sort u} \qquad \texttt{C, a} : \alpha \vdash \beta\ \texttt{a} : \texttt{Sort v}}{\texttt{C} \vdash (\forall \texttt{a} : \alpha, \beta\ \texttt{a}) : \texttt{Sort (imax u v)}}\ \textsc{Forall}$$

where `imax u 0 = 0` and `imax u (v + 1) = max u (v + 1)`.

11.2.2 Proof Irrelevance

A second difference between `Prop` and `Type` is *proof irrelevance*. It means that any two proofs of the same proposition `a` are equal:

```
lemma proof_irrel {a : Prop} (h₁ h₂ : a) :
  h₁ = h₂ :=
by refl
```

In Lean, this equality is a syntactic equality up to computation, allowing us to use 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.

An unfortunate consequence of proof irrelevance is that it prevents us from performing rule induction by pattern matching. Induction by pattern matching relies on a "measure"—a function to ℕ that assigns a size to the arguments. Without large elimination, the measure cannot be defined meaningfully. This explains why we always use the `induction` tactic for rule induction.

11.2.3 No Large Elimination

A further difference between `Prop` and `Type` is that `Prop` does not allow *large elimination*, meaning that it is generally impossible to extract information from a proof of a proposition and use it in a program (i.e., a value of a type from `Type`). After all, because of proof irrelevance, all proofs of a given proposition are equal, so they cannot carry specific information that would distinguish them.

Imagine we could extract information from proofs inside a program. We could for instance use the `match` construct in function definitions such as the following:

```
-- fails
def unsquare (i : ℤ) (hsq : ∃j, i = j * j) : ℤ :=
match hsq with
| Exists.intro j _ := j
end
```

The `unsquare` takes a square number `i` and a proof `hsq` that `i` is actually a square number and returns the number `j` before squaring, extracted from the proof. Lean raises the error

```
induction tactic failed, recursor 'Exists.dcases_on' can
only eliminate into Prop
```

but if it accepted the definition, we could derive `false` as follows. Let

```
hsq₁ := Exists.intro 3 dec_trivial
hsq₂ := Exists.intro (-3) dec_trivial
```

be two proofs of `∃j, (9 : ℤ) = j * j`. Notice that they use different witnesses for `j` (`3` versus `-3`). We then have `unsquare 9 hsq₁ = 3` and `unsquare 9 hsq₂ = -3`. However, by proof irrelevance, `hsq₁ = hsq₂`. Hence, `unsquare 9 hsq₂` equals `3`. But we already determined that it equals `-3`. This means `3 = -3`, a contradiction.

As a compromise, Lean allows *small elimination*, which eliminate only into `Prop`—whereas large elimination can eliminate into an arbitrary large universe `Sort u`. This means we can use `match` to analyze the structure of a proof, extract existential witnesses, and so on, as long as the `match` expression itself is a proof. We have seen several examples of this in Section 5.6.

As a further compromise, Lean allows large elimination for *syntactic subsingletons*: types in `Prop` for which it can be established syntactically that they have cardinality 0 or 1. These are propositions such as `false` and `a ∧ b` that can be proved in at most one way. For example, `false` has no proof, and all proofs of `a ∧ b` have the form `and.intro _ _`. (Recursively, there might be several ways to prove `a` and `b`.) More precisely, a syntactic subsingleton is an inductive definition with at most one constructor whose arguments are either `Prop` or appear as immediate arguments in the result type. When we match `h : a ∧ b` against the pattern `and.intro ha hb`, no information is leaked about `h`.

11.3 The Axiom of Choice

Lean's logic includes the axiom of choice, which makes it possible to obtain an arbitrary element of any nonempty type. Consider the following predicate:

```
inductive nonempty (α : Sort u) : Prop
| intro (val : α) : nonempty
```

The predicate states that the given type `α` has at least one element. To prove `nonempty α`, we must provide an `α` value to `nonempty.intro`:

```
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 the proof of `nonempty α`.

In Lean, the axiom of choice takes the form of a function that returns some arbitrary element of type `α` given a proof of `nonempty α`:

```
constant classical.choice {α : Sort u} :
  nonempty α → α
```

We have no way to know whether this is the same element that was used to prove `nonempty α`. It will just be an arbitrary element of `α`, thereby avoiding inconsistencies. Using the axiom of choice, we can state properties that are unprovable but whose negation is *also* unprovable—e.g., `classical.choice nat.nonempty = 42`.

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. By contrast, mathematicians generally do not mind the axiom.

Unlike proof irrelevance and large and small elimination, the axiom of choice is not built into the Lean kernel; it is only an axiom in Lean's core library, giving users the freedom to work with or without it. Lean requires us to mark definitions using the `noncomputable` keyword if they use `classical.choice` to define constants in `Type`, much in the same way as `meta` for metadefinitions (Chapter 7).

The following tools rely on `classical.choice`:

- The function `classical.some`, often called Hilbert's choice operator, can help us find a witness of `∃a : α, p a`, if we do not care which one. Its companion `classical.some_spec` gives a proof that the witness is indeed a witness.

```
classical.some : (∃a : α, p a) → α
classical.some_spec : ∀h : (∃a : α, p a), p (classical.some h)
```

- We can also derive the traditional axiom of choice:

```
classical.axiom_of_choice (α β : Type) {r : α → β → Prop} :
  (∀x : α, ∃y : β, r x y) → (∃f, ∀x, r x (f x))
```

- Using the axiom of choice, propositional extensionality (`propext`), and functional extensionality (`funext`), we can derive the law of excluded middle:

```
classical.em : ∀a : Prop, a ∨ ¬ a
```

  With the law of excluded middle, every proposition is decidable. This means that we can construct proofs based on a case distinction on whether a certain proposition is true. We introduced this technique in Chapters 4 and 5.

11.4 Subtypes

Inductive types are a very convenient definitional mechanism when they are applicable, but lots of mathematical constructions do not fit that mold. Lean provides two alternatives to cater for these: subtyping and quotienting.

Subtyping is a mechanism to create new types from existing ones. Given a predicate on the elements of a base type, the *subtype* contains only those elements of the base type that fulfill this property. More precisely, the subtype contains element–proof pairs that combine an element of the base type and a proof that it fulfills the property.

Subtyping is useful for those types that cannot be defined as an inductive type. For example, any attempt at defining the type of finite sets along the lines of

```
inductive finset (α : Type) : Type
| empty  : finset
| insert : α → finset → finset
```

is doomed to fail, because the same set can be represented in multiple ways. For example, `{1, 2}` can be represented in any of the following ways, and more:

```
finset.insert 1 (finset.insert 2 empty)
finset.insert 2 (finset.insert 1 empty)
finset.insert 1 (finset.insert 1 (finset.insert 2 empty))
```

Using a subtype, we can choose one canonical representation for each value and filter out the remaining ones. Subtypes have the syntax

{*variable* : *base-type* // *property-applied-to-variable*}

We saw an example in Section 3.6, namely, `{i : ℕ // i ≤ n}`, which consists of the natural numbers `i` fulfilling `i ≤ n`, where `n` is fixed in the context. The base type is `ℕ`, and the property is `λi, i ≤ n`. A less suggestive but perhaps less confusing syntax for the same type is `@subtype ℕ (λi, i ≤ n)`.

11.4.1 First Example: Full Binary Trees

To illustrate subtypes, we will define a type of full binary trees, building on the type `btree` of binary trees from Section 4.8. In Section 5.7.3, 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) : Type :=
{t : btree α // is_full t}
```

This is syntactic sugar for

```
def full_btree (α : Type) : Type :=
@subtype (btree α) is_full
```

where `subtype` is defined as follows:

```
inductive subtype {α : Type} (p : α → Prop) : Type
| mk : ∀x : α, p x → subtype
```

Elements of `full_btree` are essentially dependently typed pairs, where the first component is a tree `t` and the second component is a proof that `t` is full:

```
def empty_full_btree : full_btree ℕ :=
subtype.mk btree.empty is_full.empty

def full_btree_6 : full_btree ℕ :=
subtype.mk (btree.node 6 btree.empty btree.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`:

```
#reduce subtype.val full_btree_6
#check subtype.property full_btree_6
```

The most appealing aspect of subtypes is that we can *lift* the operations from the base type to the subtype instead of having to build a library from scratch, as long as they preserve the subtype property. We only need to define "wrappers" around constants from the base type. In general, defining such a wrapper given an an operation `f` on the base type involves three steps:

1. extract the base type values from the arguments to the wrapper;
2. invoke `f` on those base type values;
3. encapsulate the result using `subtype.mk`, together with a proof that the resulting base type value fulfills the subtype property.

Using this procedure, we can lift functions from `btree` to `full_btree` if they preserve the property `is_full`. To lift the `mirror` operation from type `btree → btree` to type `full_btree → full_btree`, we must

1. extract the `btree` from the argument to the wrapper;
2. invoke `mirror` on that `btree`;
3. encapsulate the result using `subtype.mk`, together with a proof that that the resulting `btree` fulfills `is_full`.

For step 3, we must extract the proof of `is_full` from the argument and use the lemma `is_full_mirror` proved in Section 5.7.3. Putting everything together, we get

```
def full_btree.mirror {α : Type} (t : full_btree α) :
  full_btree α :=
subtype.mk (mirror (subtype.val t))
  begin
    apply is_full_mirror,
    apply subtype.property t
  end
```

The input is an element `t` of the subtype `full_btree`. We decompose `t` into `subtype.val t : btree` and `subtype.property t : is_full t`. We use the `mirror` function to reverse the tree component of `t` and use the lemma `is_full_mirror`, together with the proof component of `t`, to show the condition `is_full (mirror (subtype.val t))`.

Finally, we build a pair containing the resulting tree and the proof that this tree is full using `subtype.mk`. The `subtype.mk` constructor can be seen both as a pair-like constructor and as a cast from `btree` to `full_btree`, with a second argument that guarantees that the cast is safe.

For proofs about subtypes, the following lemma is useful:

```
subtype.eq : subtype.val ?a = subtype.val ?b → ?a = ?b
```

It states that two elements of a subtype are equal if their `subtype.val` components are equal. The proofs, after all, are irrelevant in Lean, which is fortunate because we would not want to have spurious duplicates values in the subtype, differing only in their proofs. Here is how `subtype.eq` can be used to prove that the mirror image of the mirror image of a `full_btree` is the `full_btree` itself:

```
lemma full_btree.mirror_mirror {α : Type} (t : full_btree α) :
  (full_btree.mirror (full_btree.mirror t)) = t :=
begin
  apply subtype.eq,
  simp [full_btree.mirror],
  apply mirror_mirror
end
```

Applying `subtype.eq` transforms the goal into `mirror (mirror (subtype.val t)) = subtype.val t`, which matches `mirror_mirror`'s statement. The `simp` step helps understand the proof but is optional.

11.4.2 Second Example: Vectors

As a second example, consider the definition of vectors from Lean's core libraries:

```
def vector (α : Type) (n : ℕ) : Type :=
{xs : list α // list.length xs = n}
```

A vector is defined as a list of a given length. With lists, there is only one type for a list of all lengths. For vectors, we have one dedicated type for every length of a vector. The advantage of this scheme is that some operations, such as addition or scalar product of vectors, rely on the two involved vectors having the same length. We saw a morally equivalent but less practical definition of vectors in Section 4.10.

Vectors can be built from lists using `subtype.mk`:

```
def vector_123 : 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 componentwise negation of an integer vector as follows:

```
def vector.neg {n : ℕ} (v : vector ℤ n) : vector ℤ n :=
subtype.mk (list.map int.neg (subtype.val v))
  begin
    rewrite list.length_map,
    exact subtype.property v
  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 `vector.neg`:

```
lemma vector.neg_neg (n : ℕ) (v : vector ℤ n) :
  vector.neg (vector.neg v) = v :=
begin
  apply subtype.eq,
  simp [vector.neg]
end
```

The application of `subtype.eq` reduces the goal to showing the corresponding property on the underlying lists. We can then use `simp` to finish the proof.

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. Like subtyping, quotienting constructs a new type from an existing type. Unlike a subtype, a quotient type contains all of the elements of the base type—but some

elements that were different in the base type are considered equal in the quotient type. In mathematical terms, the quotient type is isomorphic to a partition of the base type.

The prerequisites to construct a quotient type are a base type τ and an equivalence relation `r : τ → τ → Prop` specifying which elements of the base type will be considered equal in the quotient. To construct the quotient type, we first need to prove that `r` as an equivalence relation on τ. A type τ equipped 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] def τ.setoid : setoid τ :=
{ r     := r,
  iseqv := ... }
```

As a side effect, this instance declaration introduces the notation `a ≈ b` for `r a b`. More importantly, we can now use the quotient type `quotient τ.setoid`.

Every element `a : τ` belongs to some element in `quotient τ.setoid`, given by

```
quotient.mk : τ → quotient τ.setoid
```

As an abbreviation for `quotient.mk a`, we can write `⟦a⟧`.

The following axiom guarantees that pairs of elements for which `r` holds are indeed equal in the quotient type:

```
quotient.sound {a b : τ} : a ≈ b → ⟦a⟧ = ⟦b⟧
```

A lemma states the converse:

```
quotient.exact {a b : τ} : ⟦a⟧ = ⟦b⟧ → a ≈ b
```

Finally, we can lift functions of type $\tau \to \upsilon$, where υ is some arbitrary type, to `quotient τ.setoid → υ` using `quotient.lift`, which has the following computation rule. Given some `f : τ → υ` such that `h : ∀a b, a ≈ b → f a = f b`, we have

```
quotient.lift f h ⟦a⟧ = f a
```

for all `a : τ`. The argument `h` corresponds to the requirement that `f` is *compatible* with ≈—i.e., it does not distinguish between ≈-equivalent arguments.

11.5.1 First Example: Integers

As an example for a quotient type, we will construct the integers. A convenient approach is to construct a quotient over pairs of natural numbers. The idea is that a pair `(p, n)` of natural numbers represents the integer `p - n`. In this way, we can represent all nonnegative integers `p` by `(p, 0)` and all negative integers `- n` by `(0, n)`. In addition, we get many more representations of the same integers; for example, `(7, 0)`, `(8, 1)`, `(9, 2)`, and `(10, 3)` all represent the integer `7`.

First, we need to register the equivalence relation that we want to use. We want two pairs (p_1, n_1) and (p_2, n_2) to be equal when $p_1 - n_1$ and $p_2 - n_2$ yield the same integer. However, the condition $p_1 - n_1 = p_2 - n_2$ does not work because subtraction on $\mathbb{N}$ is ill-behaved (e.g., `0 - 1 = 0`). Instead, we use the condition $p_1 + n_2 = p_2 + n_1$, which relies on addition.

We provide the definition of our equivalence relation, followed by a proof that it is reflexive, symmetric, and transitive:

```
@[instance] def int.rel : setoid (ℕ × ℕ) :=
{ r     :=
    λpn₁ pn₂ : ℕ × ℕ,
      prod.fst pn₁ + prod.snd pn₂ = prod.fst pn₂ + prod.snd pn₁,
  iseqv :=
    begin
      repeat { apply and.intro },
      { intro pn,
        refl },
      { intros pn₁ pn₂ h,
        rewrite h },
      { intros pn₁ pn₂ pn₃ h₁₂ h₂₃,
        apply @eq_of_add_eq_add_right _ _ _ (prod.snd pn₂),
        cc }
    end }
```

We can now write $\approx$ for the equivalence relation:

```
lemma int.rel_iff (pn₁ pn₂ : ℕ × ℕ) :
  pn₁ ≈ pn₂ ↔
  prod.fst pn₁ + prod.snd pn₂ = prod.fst pn₂ + prod.snd pn₁ :=
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 int.zero : int :=
⟦(0, 0)⟧
```

In fact, any term of the form `⟦(m, m)⟧` represents the same integer:

```
lemma int.zero_eq (m : ℕ) :
  int.zero = ⟦(m, m)⟧ :=
begin
  rewrite int.zero,
  apply quotient.sound,
  rewrite int.rel_iff,
  simp
end
```

Next, we define addition on our new integers. To define functions on a quotient type, we cannot simply define them by pattern matching, as for inductive types. Instead, we define the function on the base type first, and then we *lift* the definition to the quotient type. To achieve this, we must prove that the definition of the function `f` does not depend on the chosen representative of an equivalence class (i.e., `a ≈ b → f a = f b`). The functions `quotient.lift` (for unary functions) and `quotient.lift₂` (for binary functions) can be used to lift functions in this way.

Addition can be defined as adding the pairs of natural numbers componentwise. We then need to provide a proof that this can be lifted to a function on the

quotient by showing that $pn_1 \approx pn_1'$ and $pn_2 \approx pn_2'$ imply

```
  ⟦(prod.fst pn₁  + prod.fst pn₂  , prod.snd pn₁  + prod.snd pn₂  )⟧
= ⟦(prod.fst pn₁' + prod.fst pn₂', prod.snd pn₁' + prod.snd pn₂')⟧
```

Formally:

```
def int.add : int → int → int :=
quotient.lift₂
  (λpn₁ pn₂ : ℕ × ℕ,
     ⟦(prod.fst pn₁ + prod.fst pn₂,
       prod.snd pn₁ + prod.snd pn₂)⟧)
  begin
    intros pn₁ pn₂ pn₁' pn₂' h₁ h₂,
    apply quotient.sound,
    rewrite int.rel_iff at *,
    linarith
  end
```

The resulting function `int.add` has the intended behavior:

```
lemma int.add_eq (p₁ n₁ p₂ n₂ : ℕ) :
  int.add ⟦(p₁, n₁)⟧ ⟦(p₂, n₂)⟧ = ⟦(p₁ + p₂, n₁ + n₂)⟧ :=
by refl
```

It would be very convenient if Lean let us enter this lemma as the definition of `int.add` in the first place, presumably with the following syntax:

```
-- fails
def int.add : int → int → int
| ⟦(p₁, n₁)⟧ ⟦(p₂, n₂)⟧ := ⟦(p₁ + p₂, n₁ + n₂)⟧
```

This would indeed be a nice and intuitive syntax, but without a proof that the definition is compatible with $\approx$, we could define nonsensical functions and use them to derive `false`. For example, we could define

```
-- fails
def int.fst : int → ℕ
| ⟦(p, n)⟧ := p
```

Observe that `int.fst ⟦(0, 0)⟧ = 0` and `int.fst ⟦(1, 1)⟧ = 1`. However, since `⟦(0, 0)⟧ = ⟦(1, 1)⟧`, we get `0 = 1`, a contradiction.

We can use the characteristic lemma `int.add_eq` to prove other lemmas about `int.add`, such as

```
lemma int.add_zero (i : int) :
  int.add int.zero i = i :=
begin
  apply quotient.induction_on i,
  intro pn,
  rewrite int.zero,
  cases pn with p n,
  rewrite int.add_eq,
  apply quotient.sound,
```

```
  simp
end
```

The somewhat ill-named lemma `quotient.induction_on` is applied to perform a case distinction on `i`, replacing `i` by ⟦`pn`⟧, where `pn` is an arbitrary value of the base type $\mathbb{N} \times \mathbb{N}$. In general:

```
quotient.induction_on : ∀q : quotient ?s, (∀a, ?p ⟦a⟧) → ?p q
```

A more intuitive syntax the first proof step would have been `cases i`, but this is not supported by Lean.

11.5.2 Second Example: Unordered Pairs

Unordered pairs are pairs for which no distinction is made between the first and second components. They are usually written {*a*, *b*}. We will introduce the type `upair α` of unordered pairs over α as the quotient of pairs `(a, b)` with respect to the relation "contains the same elements as":

```
@[instance] def upair.rel (α : Type) : setoid (α × α) :=
{ r     := λab₁ ab₂ : α × α,
    ({prod.fst ab₁, prod.snd ab₁} : set α) =
    ({prod.fst ab₂, prod.snd ab₂} : set α),
  iseqv := by repeat { apply and.intro }; finish }

lemma upair.rel_iff {α : Type} (ab₁ ab₂ : α × α) :
  ab₁ ≈ ab₂ ↔
  ({prod.fst ab₁, prod.snd ab₁} : set α) =
  ({prod.fst ab₂, prod.snd ab₂} : set α) :=
by refl

def upair (α : Type) : Type :=
quotient (upair.rel α)
```

We define `upair.mk a b` to construct an unordered pair containing `a` and `b`:

```
def upair.mk {α : Type} (a b : α) : upair α :=
⟦(a, b)⟧
```

It is easy to prove that `upair.mk` is symmetric, meaning that our pairs are really unordered:

```
lemma upair.mk_symm {α : Type} (a b : α) :
  upair.mk a b = upair.mk b a :=
begin
  unfold upair.mk,
  apply quotient.sound,
  rewrite upair.rel_iff,
  apply set.insert_comm
end
```

Another representation of unordered pairs is as sets of cardinality 1 or 2. The following operation converts `upair α` values to that representation:

```
def set_of_upair {α : Type} : upair α → set α :=
quotient.lift (λab : α × α, {prod.fst ab, prod.snd ab})
  begin
    intros ab₁ ab₂ h,
    rewrite upair.rel_iff at *,
    exact h
  end
```

11.5.3 Alternative Definitions via Normalization and Subtyping

Each element of a quotient type correspond to a class of $\approx$-equivalent elements of the base type. If there exists a systematic way to obtain a canonical representative for each $\approx$-equivalence class, we can use a subtype instead of a quotient, keeping only the canonical representatives and filtering out the other elements.

Consider the quotient type `int` of integers constructed above. We observed that `(7, 0)`, `(8, 1)`, `(9, 2)`, and `(10, 3)` all represent the integer `7`, but intuitively `(7, 0)` seems preferable to the others. We will say that a pair `(p, n)` is *canonical* if `p` or `n` (or both) is `0`:

```
inductive int.is_canonical : ℕ × ℕ → Prop
| nonpos {n : ℕ} : int.is_canonical (0, n)
| nonneg {p : ℕ} : int.is_canonical (p, 0)
```

The integers are made up of the canonical pairs of natural numbers:

```
def int : Type :=
{pn : ℕ × ℕ // int.is_canonical pn}
```

Clearly, each integer can be represented in one and only one way. Operations on integers, such as addition and multiplication, must provide canonical results. Fortunately, normalizing pairs of natural numbers is easy:

```
def int.normalize : ℕ × ℕ → ℕ × ℕ
| (p, n) := if p ≥ n then (p - n, 0) else (0, n - p)

lemma int.is_canonical_normalize (pn : ℕ × ℕ) :
  int.is_canonical (int.normalize pn)
```

For unordered pairs, there is no obvious normal form, except to always put the smaller element first (or last). This requires a linear order $\leq$ on α:

```
def upair.is_canonical {α : Type} [linear_order α] :
  α × α → Prop
| (a, b) := a ≤ b

def upair (α : Type) [linear_order α] : Type :=
{ab : α × α // upair.is_canonical ab}
```

11.6 Summary of New Constructs

Declaration

`noncomputable` prefixes noncomputable declarations

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
`quotient`	function that creates a quotient type of a given setoid instance
`quotient.lift`	function that lifts a unary function to a quotient type
`quotient.lift₂`	function that lifts a binary function to a quotient type
`setoid`	type class for a type with an equivalence relation on it
`Sort u`	universe at level `u`

Notations

`{x : α // P[x]}`	subtype of all `x` in α fulfilling `P[x]`
$\approx$	equivalence relation on a setoid (used for quotienting)
`Prop`	abbreviation for `Sort 0`
`Type u`	abbreviation for `Sort (u + 1)`

Lemmas

`classical.axiom_of_choice`	traditional axiom of choice
`classical.some_spec`	characteristic property of `classical.some`
`quotient.exact`	equality on quotient type implies $\approx$ on base type
`quotient.induction_on`	case distinction on a quotient type value
`quotient.sound`	$\approx$ on base type implies equality on quotient type
`subtype.eq`	equality on base type implies equality of subtype

Chapter 12

Basic Mathematical Structures

In this chapter, we introduce definitions and proofs about basic mathematical structures such as groups, fields, and linear orders.

12.1 Type Classes over a Single Binary Operator

In mathematics, a *group* is a set G with 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 $a^{-1} \bullet a = e$.

In Lean, a type class for groups could be defined as follows:

```
@[class] structure group (α : Type) : Type :=
(mul          : α → α → α)
(one          : α)
(inv          : α → α)
(mul_assoc    : ∀a b c, mul (mul a b) c = mul a (mul b c))
(one_mul      : ∀a, mul one a = a)
(mul_left_inv : ∀a, mul (inv a) a = one)
```

However, this is not the official definition. Groups are part of a larger hierarchy of algebraic structures.

Group operations can be written multiplicatively (with operator $*$, identity element 1, and inverse element a^{-1}) or additively (with operator +, identity element 0, and inverse element $-a$). This is why Lean offers two type classes for groups: the multiplicative `group` and the additive `add_group`. They are essentially the same but use different names for their constants and properties.

Any type that satisfies the group axioms can be registered as a `group` or `add_group`. To illustrate this, we will define $\mathbb{Z}_2$, also known as $\mathbb{Z}/2\mathbb{Z}$, and register it as an `add_group`. The type $\mathbb{Z}_2$ has two elements:

```
inductive ℤ₂ : Type
| zero
| one
```

Addition is defined as follows:

```
def ℤ₂.add : ℤ₂ → ℤ₂ → ℤ₂
| ℤ₂.zero a       := a
| a       ℤ₂.zero := a
| ℤ₂.one  ℤ₂.one  := ℤ₂.zero
```

Running `#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` correspond to the binary operator $\bullet$, the identity element e, and the inverse $^{-1}$. 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 prove the redundant property `add_group.add_zero`.

The type $\mathbb{Z}_2$ can be registered as a group as follows:

```
@[instance] def ℤ₂.add_group : add_group ℤ₂ :=
{ add          := ℤ₂.add,
  add_assoc    :=
    by intros a b c; simp [(+)]; cases a; cases b; cases c;
      refl,
  zero         := ℤ₂.zero,
  zero_add     := by intro a; cases a; refl,
  add_zero     := by intro a; cases a; refl,
  neg          := λa, a,
  add_left_neg := by intro a; cases a; refl }
```

With this instance declaration, we can now use the notations `0`, `+`, and `-`:

```
#reduce ℤ₂.one + 0 - 0 - ℤ₂.one
```

Moreover, we can use any lemma that has been proved for `add_group`:

```
lemma ℤ₂.add_right_neg:
  ∀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:

Type class	Properties	Examples
`semigroup`	associativity of `*`	$\mathbb{R}$, $\mathbb{Q}$, $\mathbb{Z}$, $\mathbb{N}$
`monoid`	`semigroup` with unit `1`	$\mathbb{R}$, $\mathbb{Q}$, $\mathbb{Z}$, $\mathbb{N}$
`left_cancel_semigroup`	`semigroup` with `c * a = c * b → a = b`	
`right_cancel_semigroup`	`semigroup` with `a * c = b * c → 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 prefixed with `add_`: `add_semigroup`, `add_monoid`, `add_group`, `add_comm_semigroup`, `add_comm_monoid`, `add_comm_group`, `add_left_cancel_semigroup`, `add_right_cancel_semigroup`:

Type class	Properties	Examples
`add_semigroup`	associativity of `+`	$\mathbb{R}$, $\mathbb{Q}$, $\mathbb{Z}$, $\mathbb{N}$
`add_monoid`	`add_semigroup` with unit `0`	$\mathbb{R}$, $\mathbb{Q}$, $\mathbb{Z}$, $\mathbb{N}$
`add_left_cancel_semigroup`	`add_semigroup` with `c + a = c + b → a = b`	$\mathbb{R}$, $\mathbb{Q}$, $\mathbb{Z}$, $\mathbb{N}$
`add_right_cancel_semigroup`	`add_semigroup` with `a + c = b + c → a = b`	$\mathbb{R}$, $\mathbb{Q}$, $\mathbb{Z}$, $\mathbb{N}$
`add_group`	`add_monoid` with inverse `-`	$\mathbb{R}$, $\mathbb{Q}$, $\mathbb{Z}$

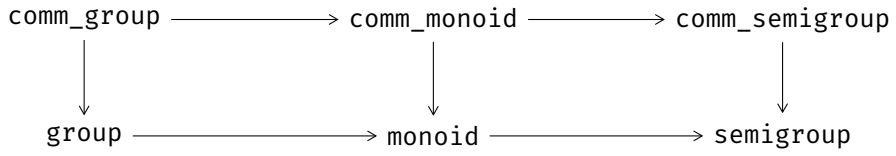
Although the additive type classes are isomorphic to their multiplicative counterparts, they are crucial when constructing algebraic structures with more than one binary operator such as rings and fields. To avoid duplicating all lemmas and definitions based on the multiplicative type classes, the copying process is automated by a tactic (`transport_multiplicative_to_additive`) and an attribute (`@[to_additive]`).

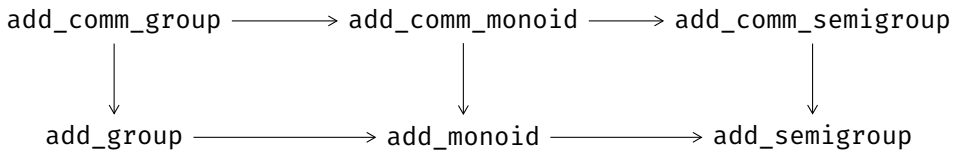
An example for an instance of `add_monoid` is the type `list` with the empty list `[]` as zero and the append operator `++` as addition:

```
@[instance] def list.add_monoid {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 continue and register `list` with `[]` and `++` as an `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."





12.2 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.
- $F \setminus \{0\}$ forms a commutative group under an operator $*$, called multiplication.
- Multiplication distributes over addition—i.e., $a * (b + c) = a * b + a * c$ for all $a, b, c \in F$.

By running `#print field`, we can display all constants and properties required by the `field` type class. Again, the class includes some redundant properties such as `left_distrib` and `right_distrib`, due to its construction.

We will now show that $\mathbb{Z}_2$ is a field by instantiating the `field` type class with it. First, we must define multiplication on $\mathbb{Z}_2$:

```
def ℤ₂.mul : ℤ₂ → ℤ₂ → ℤ₂
| ℤ₂.one  a       := a
| a       ℤ₂.one  := a
| 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] def ℤ₂.field : field ℤ₂ :=
{ one            := ℤ₂.one,
  mul            := ℤ₂.mul,
  inv            := λa, a,
  add_comm       := by intros a b; cases a; cases b; refl,
  zero_ne_one    := by finish,
  one_mul        := by intros a; cases a; refl,
  mul_one        := by intros a; cases a; refl,
  mul_inv_cancel := by intros a h; cases a; finish,
  inv_mul_cancel := by intros a h; cases a; finish,
  mul_assoc      :=
    by intros a b c; cases a; cases b; cases c; refl,
  mul_comm       := by intros a b; cases a; cases b; refl,
  left_distrib   :=
    by intros a b c; by cases a; cases b; cases c; refl,
  right_distrib  :=
    by intros a b c; by cases a; cases b; cases c; refl,
  ..ℤ₂.add_group }
```

With this declaration in place, we can now use the notations `1`, `*`, `/`, and more:

```
#reduce (1 : ℤ₂) * 0 / (0 - 1)
```

The command prints `ℤ₂.zero`. The type annotation `: ℤ₂` is necessary here to tell Lean that we want to calculate in `ℤ₂` and not in `ℕ`, the default. We can even use arbitrary numerals in `ℤ₂`. For example, the numeral `3` is interpreted as `1 + 1 + 1`, which is the same as `1` in `ℤ₂`:

```
#reduce (3 : ℤ₂)
```

The command prints `ℤ₂.one`.

Moreover, we can use the `ring` or `ring_exp` tactic to normalize terms. For example:

```
lemma ring_example (a b : ℤ₂) :
  (a + b) ^ 3 = a ^ 3 + 3 * a ^ 2 * b + 3 * a * b ^ 2 + b ^ 3 :=
by ring

lemma ring_exp_example (a b : ℤ₂) (n : ℕ):
  (a + b) ^ (2 + n) =
  (a + b) ^ n * (a ^ 2 + 2 * a * b + b ^ 2) :=
by ring_exp
```

These tactics are available for any type that is declared as a `field` or more generally as a `ring`. The distinguishing feature of `ring_exp` is that it can reason about exponents containing variables.

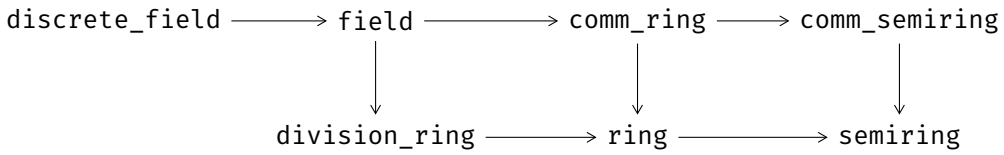
For commutative monoids and groups, there is a similar normalization tactic called `abel`:

```
lemma abel_example (a b : ℤ) :
  a + b + 0 - (b + a + a) = - a :=
by abel
```

Besides `field`, there are many more type classes for structures with two binary operators. These are the most important ones:

Type class	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 regard division as a partial function. The graph below illustrates the relationships between these type classes.



The hierarchy between `ring` and `field` is more complex than depicted and includes type classes for domains, integral domains, Euclidean rings, and more.

12.3 Coercions

When combining numbers from $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$, and $\mathbb{R}$ in the same lemmas, we may want to cast from one type to 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` and `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 ↑ or `coe`, when necessary. This coercion operator can be set up to provide implicit conversions between arbitrary types. Many coercions are already in place, including these:

- `coe : ℕ → α` casts $\mathbb{N}$ to another semiring α;
- `coe : ℤ → α` casts $\mathbb{Z}$ to another ring α;
- `coe : ℚ → α` casts $\mathbb{Q}$ to another division ring α.

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_nat_mul_neg (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 goals such as `⊢ m n : ℕ, h : ↑m = ↑n ⊢ m = n` in the proof

```
lemma norm_cast_example_1 (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

```
lemma norm_cast_example_2 (m n : ℕ) :
  (m : ℤ) + (n : ℤ) = ((m + n : ℕ) : ℤ) :=
by norm_cast
```

The `norm_cast` tactic relies on lemmas such as the following:

```
nat.cast_add : ∀a b : ℕ, ↑(a + b) = ↑a + ↑b
int.cast_add : ∀a b : ℤ, ↑(a + b) = ↑a + ↑b
rat.cast_add : ∀a b : ℚ, ↑(a + b) = ↑a + ↑b
```

12.4 Normalization Tactics

The algebraic tactics `ring`, `ring_exp`, and `abel` and the coercion tactic `norm_cast` all work by normalization: They rewrite expressions in the hope that they become syntactically equal, at which point equality is trivial to prove. Like `rewrite` and `simp`, they produce a subgoal when they make some progress but do not fully succeed.

The optional *`position`* is as for the rewriting tactics (Section 2.5).

ring and ring_exp

`ring`[`_exp`] [`at` *`position`*]

The `ring` and `ring_exp` tactics prove equalities over commutative rings and semirings (such as ℕ, ℤ, ℚ, and ℝ) by normalizing expressions and syntactically comparing the result. The `ring_exp` variant performs more aggressive normalization in the presence of variables in exponents.

abel

`abel` [`at` *`position`*]

The `abel` tactic proves equalities over additive commutative monoids and groups (such as ℕ, ℤ, ℚ, and ℝ) by normalizing expressions and syntactically comparing the result.

norm_cast

`norm_cast` [`at` *`position`*]

The `norm_cast` tactic moves coercions towards the inside of expressions, as a form of simplification.

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 the following definition, based on the binary trees we introduced in Section 4.8:

```
def list.elems : btree ℕ → list ℕ
| btree.empty        := []
| (btree.node a l r) := a :: list.elems l ++ list.elems r
```

This function returns a list of all the elements occurring in the tree. It traverses the tree depth first, from left to right. But for some applications, we might not care 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 multiset.elems : btree ℕ → multiset ℕ
| btree.empty        := ∅
| (btree.node a l r) :=
  {a} ∪ (multiset.elems l ∪ multiset.elems r)
```

Using this definition, we can prove that `multiset.elems t = multiset.elems (mirror t)`, whereas `list.elems t = list.elems (mirror t)` does not hold.

For some applications, we might want to go a step further and ignore not only the order but also how often each element occurs in the tree, distinguishing only between occurrence and nonoccurrence. This is where finite sets, or *finsets*, come into play. On finsets, we have `{3, 2, 1, 2} = {1, 2, 3}`. Finsets are defined as the subtype of multisets that do not contain any repeated elements. (Another possible definition would have been as the subtype of sets that are finite.) We can redo the definition above using finsets as follows:

```
def finset.elems : btree ℕ → finset ℕ
| btree.empty        := ∅
| (btree.node a l r) := {a} ∪ (finset.elems l ∪ finset.elems r)
```

For all three of these structures, Lean offers sum and product operators to add or multiply all of the elements:

```
#eval list.sum [2, 3, 4]                          -- result: 9
#eval multiset.sum ({2, 3, 4} : multiset ℕ)       -- result: 9
#eval finset.sum ({2, 3, 4} : finset ℕ) (λn, n)   -- result: 9

#eval list.prod [2, 3, 4]                         -- result: 24
#eval multiset.prod ({2, 3, 4} : multiset ℕ)      -- result: 24
#eval finset.prod ({2, 3, 4} : finset ℕ) (λn, n)  -- 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 `λn, n` 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`, they additionally require an instance declaration for `add_comm_monoid` or `comm_monoid` because the result cannot depend on the order of adding or multiplying the elements.

Lean's `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 order* (or *total order*) is a binary relation $\leq$ such that for all a, b, and c, the following properties hold:

- Reflexivity: $a \leq a$.
- Transitivity: If $a \leq b$ and $b \leq c$, then $a \leq c$.
- Antisymmetry: If $a \leq b$ and $b \leq a$, then $a = b$.
- Totality: $a \leq b$ or $b \leq a$.

If a relation has the first three properties, it is a *partial order*. An example is the subset relation $\subseteq$ on sets, finite sets, or multisets. If a relation has the first two properties, it is a *preorder*. An example is comparing lists by their length.

In Lean, there are type classes for the different kinds of orders: `linear_order`, `partial_order`, and `preorder`. The `preorder` 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` 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
```

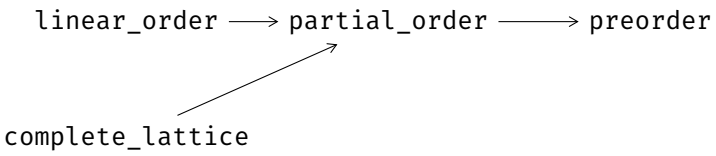
We can declare the preorder on `list α` that compares lists by their length as follows:

```
@[instance] def list.length.preord {α : Type} :
  preorder (list α) :=
{ le      := λxs ys, list.length xs ≤ list.length ys,
  le_refl  := by intro xs; exact nat.le_refl _,
  le_trans := by intros xs ys zs; exact nat.le_trans }
```

This type class instance gives us access to the infix syntax $\leq$ and to the corresponding relations $\geq$, <, and >:

```
lemma list.length.preord_example {α : Type} (c : α) :
  [c] > [] :=
dec_trivial
```

Complete lattices, which we discussed in Chapter 10, are formalized as another type class, `complete_lattice`, which inherits from `partial_order`. The arrows in the following graphical overview mean "inherits all properties from."



Finally, Lean provides type classes that combine orders and algebraic structures: `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., `c + a` $\leq$ `c + b` $\to$ `a` $\leq$ `b`).

12.7 Summary of New Constructs

Notation

↑	coercion operator `coe`

Tactics

`abel`	normalizes terms of commutative monoids and groups
`norm_cast`	normalizes coercions
`ring`	normalizes ring expressions
`ring_exp`	normalizes ring expressions including exponents

Chapter 13

Rational and Real Numbers

We have seen how the natural numbers $\mathbb{N}$ can be defined as an inductive type and how the integers $\mathbb{Z}$ can be defined as a quotient over $\mathbb{N} \times \mathbb{N}$. In this chapter, we review the construction of the rational numbers $\mathbb{Q}$ and the real numbers $\mathbb{R}$. The tools used for these constructions are inductive types, subtypes, and quotients.

This procedure can be followed to construct types with specific properties:

1. Create a new type that is large enough to represent all elements, but not necessarily in a unique manner.
2. Quotient this representation, equating elements that should be equal.
3. Define operators on the quotient type by lifting functions from the base type, and prove that they are compatible with the quotient relation.

We have used this approach to construct $\mathbb{Z}$ in Section 11.5.1. It can be employed 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, where $d \neq 0$:

```
structure fraction : Type :=
(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: $n_1 * d_2 = n_2 * d_1$. To construct the quotient over the type `fraction` with respect to this relation, we show that the relation is an equivalence relation. This is achieved by declaring `fraction` an instance of the `setoid` type class:

```
namespace fraction

@[instance] def setoid : setoid fraction :=
```

```
{ r     := λa b : fraction, num a * denom b = num b * denom a,
  iseqv :=
    begin
      repeat { apply and.intro },
      { intros a; refl },
      { intros a b h; cc },
      { intros a b c eq_ab eq_bc,
        apply eq_of_mul_eq_mul_right (denom_ne_zero b),
        cc }
    end }

lemma setoid_iff (a b : fraction) :
  a ≈ b ↔ num a * denom b = num b * denom a :=
by refl

end fraction
```

Then we can define the type of rationals as the quotient over this setoid:

```
def rat : Type :=
quotient fraction.setoid
```

To define zero, one, addition, multiplication, and other operations, we first define them on the `fraction` type. To add two fractions, we convert 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] def has_add : has_add fraction :=
{ add := λa b : fraction,
    { num           := num a * denom b + num b * denom a,
      denom         := denom a * denom b,
      denom_ne_zero :=
        by apply mul_ne_zero; exact 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 `rat`, we must prove them compatible with ≈:

```
namespace fraction

@[simp] lemma add_num (a b : fraction) :
  num (a + b) = num a * denom b + num b * denom a :=
by refl

@[simp] lemma add_denom (a b : fraction) :
  denom (a + b) = denom a * denom b :=
by refl

lemma add_equiv_add {a a' b b' : fraction} (ha : a ≈ a')
    (hb : b ≈ b') :
  a + b ≈ a' + b' :=
```

```
  begin
    simp [setoid_iff] at *,
    calc  (num a * denom b + num b * denom a)
            * (denom a' * denom b')
        = num a * denom a' * denom b * denom b'
            + num b * denom b' * denom a * denom a' :
      by simp [add_mul]; ac_refl
    ... = num a' * denom a * denom b * denom b'
            + num b' * denom b * denom a * denom a' :
      by simp [*]
    ... = (num a' * denom b' + num b' * denom a')
            * (denom a * denom b) :
      by simp [add_mul]; ac_refl
  end

  end fraction
```

(The `ac_refl` tactic is a variant of `refl` that reasons up to associativity and commutativity, much like `cc`.) Then we can use `quotient.lift`$_2$ to define the `rat` operations, and we can instantiate the relevant syntactic type classes, such as

```
  namespace rat

  @[instance] def has_add : has_add rat :=
  { add := quotient.lift₂ (λa b : fraction, ⟦a + b⟧)
      begin
        intros a b a' b' ha hb,
        apply quotient.sound,
        exact fraction.add_equiv_add ha hb
      end }

  end rat
```

From here, we can proceed and prove all the properties needed to make `rat` an instance of `field` or, if we are more ambitious, `discrete_linear_ordered_field`.

Alternative Definitions of the Rational Numbers There are at least two more approaches to construct the rational numbers. The first approach is essentially the one used in `mathlib`. The type `rat` is defined as a subtype of `fraction`, with the requirement that the numerator and the denominator have no common divisors except 1 and -1:

```
def rat.is_canonical (a : fraction) : Prop :=
nat.coprime (int.nat_abs (fraction.num a))
  (int.nat_abs (fraction.denom a))

def rat := {a : fraction // rat.is_canonical a}
```

We discussed this general strategy in Section 11.5.3. In this way, no quotient is required, computation is more efficient, and more properties are equalities up to computation. A disadvantage is that function definitions become more complicated, due to the need to normalize fractions.

Another alternative 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 rat.rel : pre_rat → pre_rat → Prop
| add_congr {a b c d : pre_rat} :
  rat.rel a b → rat.rel c d →
  rat.rel (pre_rat.add a c) (pre_rat.add b d)
| add_assoc {a b c : pre_rat} :
  rat.rel (pre_rat.add a (pre_rat.add b c))
    (pre_rat.add (pre_rat.add a b) c)
| zero_add {a : pre_rat} :
  rat.rel (pre_rat.add pre_rat.zero a) a
-- etc.

def rat : Type :=
quot rat.rel
```

This construction does not require $\mathbb{Z}$ and that it can easily be registered as a `field`. Moreover, the same approach can be used for other algebraic constructions, such as free monoids and free groups. A drawback is that the definition of orders and lemmas about them become more complicated.

13.2 Real Numbers

Some sequences of rational numbers seem to converge because the numbers in the sequence get closer and closer to each other, and yet do not converge to a rational number. The sequence

$$
\begin{aligned}
a_0 &= 1 \\
a_1 &= 1.4 \\
a_2 &= 1.41 \\
a_3 &= 1.414 \\
a_4 &= 1.4142 \\
a_5 &= 1.41421 \\
a_6 &= 1.414213 \\
a_7 &= 1.4142135 \\
&\vdots
\end{aligned}
$$

where a_n is the largest number with n digits after the decimal point such that $a_n^2 < 2$, is such a sequence. It seems to converge because each a_n is at most 10^{-n} away from any of the following numbers, but the limit is $\sqrt{2} \notin \mathbb{Q}$. In that sense, the rational numbers are incomplete, and the reals are their *completion*. To construct the reals, we need to fill in the gaps that are revealed by these sequences that seem to converge but do not.

Cauchy sequences capture the notion of a sequence that seems to converge. A sequence $a_0, a_1, \ldots$ is *Cauchy* if for any $\varepsilon > 0$, there exists an $N \in \mathbb{N}$ such that for all $m \geq N$, we have $|a_N - a_m| < \varepsilon$. In other words, no matter how small we choose ε, we can always find a point in the sequence from which all following numbers deviate by less than ε.

We formalize sequences of rational numbers as functions `f : ℕ → ℚ` and denote the absolute value | | by `abs`. This yields the following Lean definition of Cauchy sequences:

```
def is_cau_seq (f : ℕ → ℚ) : Prop :=
∀ε > 0, ∃N, ∀m ≥ N, abs (f N - f m) < ε
```

Not every sequence is a Cauchy sequence:

```
lemma id_not_cau_seq :
  ¬ is_cau_seq (λn : ℕ, (n : ℚ)) :=
begin
  rewrite is_cau_seq,
  intro h,
  cases h 1 zero_lt_one with i hi,
  have hi_succi :=
    hi (i + 1) (by simp),
  simp at hi_succi,
  linarith
end
```

We define a type of Cauchy sequences as a subtype:

```
def cau_seq : Type :=
{f : ℕ → ℚ // is_cau_seq f}
```

It will be convenient to have an auxiliary function that extracts the actual sequence from a `cau_seq`:

```
def seq_of (f : cau_seq) : ℕ → ℚ :=
subtype.val f
```

The basic idea of the construction is to represent the real numbers by Cauchy sequences. Each Cauchy sequence represents the real number that is its limit; for example, the sequence $a_n = 1/n$ represents the real number 0, and the sequence 1, 1.4, 1.41, ... represents the real number $\sqrt{2}$.

Two different Cauchy sequences can represent the same real number; for example, the sequence $a_n = 1/n$ and the constant sequence $b_n = 0$ both represent 0. Therefore, we need to take the quotient over sequences representing the same real number. Two sequences represent the same real number when their difference converges to zero:

```
namespace cau_seq
```

```
@[instance] def setoid : setoid cau_seq :=
{ r     := λf g : cau_seq,
    ∀ε > 0, ∃N, ∀m ≥ N, abs (seq_of f m - seq_of g m) < ε,
  iseqv :=
    begin
      apply and.intro,
      { intros f ε hε,
        apply exists.intro 0,
        finish },
      apply and.intro,
      { intros f g hfg ε hε,
        cases hfg ε hε with N hN,
        apply exists.intro N,
        intros m hm,
        rewrite abs_sub,
        apply hN m hm },
      { intros f g h hfg hgh ε hε,
        cases hfg (ε / 2) (half_pos hε) with N₁ hN₁,
        cases hgh (ε / 2) (half_pos hε) with N₂ hN₂,
        apply exists.intro (max N₁ N₂),
        intros m hm,
        calc  abs (seq_of f m - seq_of h m)
            ≤ abs (seq_of f m - seq_of g m)
              + abs (seq_of g m - seq_of h m) :
          by apply abs_sub_le
        ... < ε / 2 + ε / 2 :
          add_lt_add (hN₁ m (le_of_max_le_left hm))
            (hN₂ m (le_of_max_le_right hm))
        ... = ε :
          by simp }
    end }

lemma setoid_iff (f g : cau_seq) :
  f ≈ g ↔
  ∀ε > 0, ∃N, ∀m ≥ N, abs (seq_of f m - seq_of g m) < ε :=
by refl

end cau_seq
```

Using this `setoid` instance, we can now define the real numbers:

```
def real : Type :=
quotient cau_seq.setoid
```

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 `real` afterwards. For the constants `0` and `1`, we can define them simply as constant sequences. Any constant sequence is a Cauchy sequence:

```
namespace cau_seq
```

```
def const (q : ℚ) : cau_seq :=
subtype.mk (λ_ : ℕ, q)
  (by rewrite is_cau_seq; intros ε hε; finish)
```

We can declare `real` instances of the syntactic type classes `has_zero` and `has_one` as follows:

```
@[instance] def has_zero : has_zero real :=
{ zero := ⟦cau_seq.const 0⟧ }

@[instance] def has_one : has_one 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] def has_add : has_add cau_seq :=
{ add := λf g : cau_seq,
    subtype.mk (λn : ℕ, seq_of f n + seq_of g 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.

Next, we need to show that this addition is compatible with $\approx$:

```
lemma add_equiv_add {f f' g g' : cau_seq} (hf : f ≈ f')
    (hg : g ≈ g') :
  f + g ≈ f' + g' :=
begin
  intros ε₀ hε₀,
  simp [setoid_iff],
  cases hf (ε₀ / 2) (half_pos hε₀) with Nf hNf,
  cases hg (ε₀ / 2) (half_pos hε₀) with Ng hNg,
  apply exists.intro (max Nf Ng),
  intros m hm,
  calc  abs (seq_of (f + g) m - seq_of (f' + g') m)
      = abs ((seq_of f m + seq_of g m)
          - (seq_of f' m + seq_of g' m)) :
    by refl
  ... = abs ((seq_of f m - seq_of f' m)
          + (seq_of g m - seq_of g' m)) :
    by simp
  ... ≤ abs (seq_of f m - seq_of f' m)
      + abs (seq_of g m - seq_of g' m) :
    by apply abs_add
  ... < ε₀ / 2 + ε₀ / 2 :
    add_lt_add (hNf m (le_of_max_le_left hm))
      (hNg m (le_of_max_le_right hm))
  ... = ε₀ :
    by simp
end

end cau_seq
```

To prove that `f + g ≈ f' + g'`, we are given an `ε₀ > 0` and must show that there exists a number `N` such that

```
∀m, m ≥ N → abs (subtype.val (f + g) m - subtype.val (f' + g') m) < ε₀
```

To obtain such an `N`, we use `f ≈ f'` and `g ≈ g'`. The equivalence `f ≈ f'` gives us for any `ε > 0` a number `Nf` such that `abs (seq_of f m - seq_of f' m) < ε` for all `m ≥ Nf`. The fact `g ≈ g'` gives us a number `Ng` with a similar property. For the calculations to work out in the end, we take the numbers `Nf` and `Ng` for `ε := ε₀ / 2`. Then we choose `N` to be the maximum of `Nf` and `Ng`, so that we get the inequalities for any `m ≥ N`. The `calc` block at the end of the proof establishes that, for all `m ≥ N`,

```
abs (seq_of (f + g) m - seq_of (f' + g') m) < ε₀
```

Using this lemma, we can define addition on `real`:

```
namespace real

@[instance] def has_add : has_add real :=
{ add := quotient.lift₂ (λa b : cau_seq, ⟦a + b⟧)
    begin
      intros a b a' b' ha hb,
      apply quotient.sound,
      exact cau_seq.add_equiv_add ha hb,
    end }

end real
```

Multiplication can be defined similarly.

In summary, real numbers are defined as a quotient over Cauchy sequences, which are in turn defined as a subtype of $\mathbb{N} \to \mathbb{Q}$.

Alternative Definitions of the Real Numbers In `mathlib`, the construction of the real numbers is essentially as described above. Only some definitions are stated in a more general fashion to allow construction of other algebraic structures, such as the *p*-adic numbers [18].

Alternatively, the real numbers can be defined using Dedekind cuts. Essentially, a number `r : ℝ` is then represented as `{x : ℚ | x < r}`.

Another alternative definition of the reals is to define them using binary sequences $\mathbb{N} \to$ `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.

13.3 Summary of New Constructs

Tactic

`ac_refl`	proves equalities `l = r` up to associativity and commutativity

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. https://leanprover.github.io/papers/tactic.pdf.

[6] G. Gonthier. Formal proof—The Four-Color Theorem. *Notices AMS*, 55(11):1382–1393, 2008. https://www.ams.org/notices/200811/tx081101382p.pdf.

[7] G. Gonthier, A. Asperti, J. Avigad, Y. Bertot, C. Cohen, F. Garillot, S. L. Roux, A. Mahboubi, R. O'Connor, S. O. Biha, I. Pasca, L. Rideau, A. Solovyev, E. Tassi, and L. Théry. A machine-checked proof of the Odd Order Theorem. In S. Blazy, C. Paulin-Mohring, and D. Pichardie, editors, *ITP 2013*, volume 7998 of *Lecture Notes in Computer Science*, pages 163–179. Springer, 2013. https://hal.inria.fr/hal-00816699/document.

[8] K. Gopinathan and I. Sergey. Certifying certainty and uncertainty in approximate membership query structures. In S. Lahiri and C. Wang, editors, *CAV 2020*, Lecture Notes in Computer Science. Springer, 2020. https://ilyasergey.net/papers/ceramist-draft.pdf.

[9] R. Gu, Z. Shao, H. Chen, X. N. Wu, J. Kim, V. Sjöberg, and D. Costanzo. CertiKOS: An extensible architecture for building certified concurrent OS kernels. In K. Keeton and T. Roscoe, editors, *OSDI 2016*, pages 653–669. USENIX Association, 2016. https://www.usenix.org/system/files/conference/osdi16/osdi16-gu.pdf.

[10] T. C. Hales, M. Adams, G. Bauer, D. T. Dang, J. Harrison, T. L. Hoang, C. Kaliszyk, V. Magron, S. McLaughlin, T. T. Nguyen, T. Q. Nguyen, T. Nipkow, S. Obua, J. Pleso, J. Rute, A. Solovyev, A. H. T. Ta, T. N. Tran, D. T. Trieu, J. Urban, K. K. Vu, and R. Zumkeller. A formal proof of the Kepler conjecture. *CoRR*, abs/1501.02155, 2015. http://arxiv.org/abs/1501.02155.

[11] J. Harrison. Formal verification at Intel. In *LICS 2003*, pages 45–54. IEEE Computer Society, 2003. https://ieeexplore.ieee.org/document/1210044.

[12] J. Harrison, J. Urban, and F. Wiedijk. History of interactive theorem proving. In J. H. Siekmann, editor, *Computational Logic*, volume 9 of *Handbook of the History of Logic*, pages 135–214. Elsevier, 2014. https://www.cl.cam.ac.uk/~jrh13/papers/joerg.pdf.

[13] J. R. Hindley. The principal type-scheme of an object in combinatory logic. *Trans. Amer. Math. Soc.*, 146:29–60, 1969. https://www.jstor.org/stable/pdf/1995158.pdf?casa_token=zeSppXjXzUsAAAAA:fzvl6NKVjcL0iVXjyrMSbh_bq0sxtY4iPeoX5NMC6FvlOdMDxYG1p-aIiqrc6P2Q7UTaT46PVHByMTWrpox4SKwOhbcxjLIm-FxWRDx-FtdyWwHa0ueG.

[14] G. Klein, J. Andronick, K. Elphinstone, G. Heiser, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal verification of an operating-system kernel. *Commun. ACM*, 53(6):107–115, 2010. https://www.researchgate.net/profile/Gernot_Heiser/publication/220910193_SeL4_Formal_verification_of_an_OS_kernel/links/09e4150f00292a0329000000/SeL4-Formal-verification-of-an-OS-kernel.pdf.

[15] D. E. Knuth. *The TEXbook*. Addison-Wesley, 1986. http://www.ctex.org/documents/shredder/src/texbook.pdf.

[16] R. Kumar, M. O. Myreen, M. Norrish, and S. Owens. CakeML: A verified implementation of ML. In S. Jagannathan and P. Sewell, editors, *POPL 2014*, pages 179–192. ACM, 2014. https://cakeml.org/popl14.pdf.

[17] X. Leroy. A formally verified compiler back-end. *J. Automated Reasoning*, 43(4):363–446, 2009. https://arxiv.org/pdf/0902.2137.pdf.

[18] R. Y. Lewis. A formal proof of Hensel's lemma over the *p*-adic integers. In A. Mahboubi and M. O. Myreen, editors, *CPP 2019*, pages 15–26. ACM, 2019. https://arxiv.org/pdf/1909.11342.pdf.

[19] M. Lipovaca. *Learn You a Haskell for Great Good!: A Beginner's Guide*. No Starch Press, 1st edition, 2011. http://learnyouahaskell.com/.

[20] A. Lochbihler. Verifying a compiler for Java threads. In A. D. Gordon, editor, *ESOP 2010*, volume 6012 of *Lecture Notes in Computer Science*, pages 427–447. Springer, 2010. https://link.springer.com/content/pdf/10.1007/978-3-642-11957-6_23.pdf.

[21] The mathlib Community. The Lean mathematical library. In J. Blanchette and C. Hriţcu, editors, *CPP 2020*, pages 367–381. ACM, 2020. https://arxiv.org/pdf/1910.09336.pdf.

[22] R. Milner. A theory of type polymorphism in programming. *J. Comput. Syst. Sci.*, 17(3):348–375, 1978. https://www.sciencedirect.com/science/article/pii/

0022000078900144/pdf?md5=cdcf7cdb7cfd2e1e4237f4f779ca0df7&pid=1-s2.0-0022000078900144-main.pdf&_valck=1.

[23] R. Nederpelt and H. Geuvers. *Type Theory and Formal Proof: An Introduction*. Cambridge University Press, 2014.

[24] T. Nipkow and G. Klein. *Concrete Semantics—With Isabelle/HOL*. Springer, 2014. http://www.concrete-semantics.org/concrete-semantics.pdf.

[25] B. O'Sullivan, D. Stewart, and J. Goerzen. *Real World Haskell*. O'Reilly, 2008. http://book.realworldhaskell.org/read/.

[26] A. J. Perlis. Epigrams on programming. *SIGPLAN Notices*, 17(9):7–13, 1982. https://dl.acm.org/doi/pdf/10.1145/947955.1083808?casa_token=Z_grmOD_yAYAAAAA:dvMu7thQc-8XmGJCZ1ARQ24_XB1qel3M1jYUmBPAKfuBaZrryxmKiKValMrwpunJ0dv9vhr2kDvMxA.

[27] B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[28] F. van Raamsdonk. *Logical Verification: Course Notes*. 2011. https://www.cs.vu.nl/~jbe248/lv2017/notes.pdf.

[29] D. M. Russinoff. A mechanically checked proof of correctness of the AMD K5 floating point square root microcode. *Formal Methods in System Design*, 14(1):75–125, 1999. https://link.springer.com/content/pdf/10.1023/A:1008669628911.pdf.

[30] D. Selsam and L. de Moura. Congruence closure in intensional type theory. In N. Olivetti and A. Tiwari, editors, *IJCAR 2016*, volume 9706 of *Lecture Notes in Computer Science*, pages 99–115. Springer, 2016. https://arxiv.org/pdf/1701.04391.pdf.

[31] C. Watt. Mechanising and verifying the WebAssembly specification. In J. Andronick and A. P. Felty, editors, *CPP 2018*, pages 53–65. ACM, 2018. https://www.cl.cam.ac.uk/~caw77/papers/mechanising-and-verifying-the-webassembly-specification.pdf.