Functional Programming in Lean

by David Thrane Christiansen

This is a free book on using Lean as a programming language. All code samples are tested with Lean release `4.18.0`.

Release history

April, 2025

The book has been extensively updated and now describes Lean version 4.18.

January, 2024

This is a minor bugfix release that fixes a regression in an example program.

October, 2023

In this first maintenance release, a number of smaller issues were fixed and the text was brought up to date with the latest release of Lean.

May, 2023

The book is now complete! Compared to the April pre-release, many small details have been improved and minor mistakes have been fixed.

April, 2023

This release adds an interlude on writing proofs with tactics as well as a final chapter that combines discussion of performance and cost models with proofs of termination and program equivalence. This is the last release prior to the final release.

March, 2023

This release adds a chapter on programming with dependent types and indexed families.

January, 2023

This release adds a chapter on monad transformers that includes a description of the imperative features that are available in `do`-notation.

December, 2022

This release adds a chapter on applicative functors that additionally describes structures and type classes in more detail. This is accompanied with improvements to the description of monads. The December 2022 release was delayed until January 2023 due to winter holidays.

November, 2022

This release adds a chapter on programming with monads. Additionally, the example of using JSON in the coercions section has been updated to include the complete code.

October, 2022

This release completes the chapter on type classes. In addition, a short interlude introducing propositions, proofs, and tactics has been added just before the chapter on type classes, because a small amount of familiarity with the concepts helps to understand some of the standard library type classes.

September, 2022

This release adds the first half of a chapter on type classes, which are Lean's mechanism for overloading operators and an important means of organizing code and structuring libraries. Additionally, the second chapter has been updated to account for changes in Lean's stream

API.

August, 2022

This third public release adds a second chapter, which describes compiling and running programs along with Lean's model for side effects.

July, 2022

The second public release completes the first chapter.

June, 2022

This was the first public release, consisting of an introduction and part of the first chapter.

About the Author

David Thrane Christiansen has been using functional languages for twenty years, and dependent types for ten. Together with Daniel P. Friedman, he wrote *The Little Typer*, an introduction to the key ideas of dependent type theory. He has a Ph.D. from the IT University of Copenhagen. During his studies, he was a major contributor to the first version of the Idris language. Since leaving academia, he has worked as a software developer at Galois in Portland, Oregon and Deon Digital in Copenhagen, Denmark, and he was the Executive Director of the Haskell Foundation. At the time of writing, he is employed at the Lean Focused Research Organization working full-time on Lean.

License

The original version of the book was written by David Thrane Christiansen on contract to Microsoft Corporation, who generously released it under a Creative Commons Attribution 4.0 International License. The current version has been modified by the author from the original version to account for changes in newer versions of Lean. A detailed account of the changes can be found in the book's source code repository.

Lean is an interactive theorem prover developed at Microsoft Research, based on dependent type theory. Dependent type theory unites the worlds of programs and proofs; thus, Lean is also a programming language. Lean takes its dual nature seriously, and it is designed to be suitable for use as a general-purpose programming language—Lean is even implemented in itself. This book is about writing programs in Lean.

When viewed as a programming language, Lean is a strict pure functional language with dependent types. A large part of learning to program with Lean consists of learning how each of these attributes affects the way programs are written, and how to think like a functional programmer. *Strictness* means that function calls in Lean work similarly to the way they do in most languages: the arguments are fully computed before the function's body begins running. *Purity* means that Lean programs cannot have side effects such as modifying locations in memory, sending emails, or deleting files without the program's type saying so. Lean is a *functional* language in the sense that functions are first-class values like any other and that the execution model is inspired by the evaluation of mathematical expressions. *Dependent types*, which are the most unusual feature of Lean, make types into a first-class part of the language, allowing types to contain programs and programs to compute types.

This book is intended for programmers who want to learn Lean, but who have not necessarily used a functional programming language before. Familiarity with functional languages such as Haskell, OCaml, or F# is not required. On the other hand, this book does assume knowledge of concepts like loops, functions, and data structures that are common to most programming languages. While this book is intended to be a good first book on functional programming, it is not a good first book on programming in general.

Mathematicians who are using Lean as a proof assistant will likely need to write custom proof automation tools at some point. This book is also for them. As these tools become more sophisticated, they begin to resemble programs in functional languages, but most working mathematicians are trained in languages like Python and Mathematica. This book can help bridge the gap, empowering more mathematicians to write maintainable and understandable proof automation tools.

This book is intended to be read linearly, from the beginning to the end. Concepts are introduced one at a time, and later sections assume familiarity with earlier sections. Sometimes, later chapters will go into depth on a topic that was only briefly addressed earlier on. Some sections of the book contain exercises. These are worth doing, in order to cement your understanding of the section. It is also useful to explore Lean as you read the book, finding creative new ways to use what you have learned.

Getting Lean

Before writing and running programs written in Lean, you'll need to set up Lean on your own computer. The Lean tooling consists of the following:

- `elan` manages the Lean compiler toolchains, similarly to `rustup` or `ghcup`.
- `lake` builds Lean packages and their dependencies, similarly to `cargo`, `make`, or Gradle.
- `lean` type checks and compiles individual Lean files as well as providing information to programmer tools about files that are currently being written. Normally, `lean` is invoked by other tools rather than directly by users.
- Plugins for editors, such as Visual Studio Code or Emacs, that communicate with `lean` and present its information conveniently.

Please refer to the Lean manual for up-to-date instructions for installing Lean.

Typographical Conventions

Code examples that are provided to Lean as *input* are formatted like this:

```
def add1 (n : Nat) : Nat := n + 1

#eval add1 7
```

The last line above (beginning with `#eval`) is a command that instructs Lean to calculate an answer. Lean's replies are formatted like this:

```
8
```

Error messages returned by Lean are formatted like this:

```
application type mismatch
  add1 "seven"
argument
  "seven"
has type
  String : Type
but is expected to have type
  Nat : Type
```

Warnings are formatted like this:

```
declaration uses 'sorry'
```

Unicode

Idiomatic Lean code makes use of a variety of Unicode characters that are not part of ASCII. For instance, Greek letters like `α` and `β` and the arrow `→` both occur in the first chapter of this book. This allows Lean code to more closely resemble ordinary mathematical notation.

With the default Lean settings, both Visual Studio Code and Emacs allow these characters to be typed with a backslash (`\`) followed by a name. For example, to enter `α` , type `\alpha` . To find out how to type a character in Visual Studio Code, point the mouse at it and look at the tooltip. In Emacs, use `C-c C-k` with point on the character in question.

Acknowledgments

This free online book was made possible by the generous support of Microsoft Research, who paid for it to be written and given away. During the process of writing, they made the expertise of the Lean development team available to both answer my questions and make Lean easier to use. In particular, Leonardo de Moura initiated the project and helped me get started, Chris Lovett set up the CI and deployment automation and provided great feedback as a test reader, Gabriel Ebner provided technical reviews, Sarah Smith kept the administrative side working well, and Vanessa Rodriguez helped me diagnose a tricky interaction between the source-code highlighting library and certain versions of Safari on iOS.

Writing this book has taken up many hours outside of normal working hours. My wife Ellie Thrane Christiansen has taken on a larger than usual share of running the family, and this book could not exist if she had not done so. An extra day of work each week has not been easy for my family—thank you for your patience and support while I was writing.

The online community surrounding Lean provided enthusiastic support for the project, both technical and emotional. In particular, Sebastian Ullrich provided key help when I was learning Lean's metaprogramming system in order to write the supporting code that allowed the text of error messages to be both checked in CI and easily included in the book itself. Within hours of posting a new revision, excited readers would be finding mistakes, providing suggestions, and showering me with kindness. In particular, I'd like to thank Arien Malec, Asta Halkjær From, Bulhwi Cha, Craig Stuntz, Daniel Fabian, Evgenia Karunus, eyelash, Floris van Doorn, František Silváši, Henrik Böving, Ian Young, Jeremy Salwen, Jireh Loreaux, Kevin Buzzard, Lars Ericson, Liu Yuxi, Mac Malone, Malcolm Langfield, Mario Carneiro, Newell Jensen, Patrick Massot, Paul Chisholm, Pietro Monticone, Tomas Puverle, Yaël Dillies, Zhiyuan Bao, and Zyad Hassan for their many suggestions, both stylistic and technical.

According to tradition, a programming language should be introduced by compiling and running a program that displays `"Hello, world!"` on the console. This simple program ensures that the language tooling is installed correctly and that the programmer is able to run the compiled code.

Since the 1970s, however, programming has changed. Today, compilers are typically integrated into text editors, and the programming environment offers feedback as the program is written. Lean is no exception: it implements an extended version of the Language Server Protocol that allows it to communicate with a text editor and provide feedback as the user types.

Languages as varied as Python, Haskell, and JavaScript offer a read-eval-print-loop (REPL), also known as an interactive toplevel or a browser console, in which expressions or statements can be entered. The language then computes and displays the result of the user's input. Lean, on the other hand, integrates these features into the interaction with the editor, providing commands that cause the text editor to display feedback integrated into the program text itself. This chapter provides a short introduction to interacting with Lean in an editor, while Hello, World! describes how to use Lean traditionally from the command line in batch mode.

It is best if you read this book with Lean open in your editor, following along and typing in each example. Please play with the examples, and see what happens!

Evaluating Expressions

The most important thing to understand as a programmer learning Lean is how evaluation works. Evaluation is the process of finding the value of an expression, just as one does in arithmetic. For instance, the value of 15 - 6 is 9 and the value of 2 × (3 + 1) is 8. To find the value of the latter expression, 3 + 1 is first replaced by 4, yielding 2 × 4, which itself can be reduced to 8. Sometimes, mathematical expressions contain variables: the value of *x* + 1 cannot be computed until we know what the value of *x* is. In Lean, programs are first and foremost expressions, and the primary way to think about computation is as evaluating expressions to find their values.

Most programming languages are *imperative*, where a program consists of a series of statements that should be carried out in order to find the program's result. Programs have access to mutable memory, so the value referred to by a variable can change over time. In addition to mutable state, programs may have other side effects, such as deleting files, making outgoing network connections, throwing or catching exceptions, and reading data from a database. "Side effects" is essentially a catch-all term for describing things that may happen in a program that don't follow the model of evaluating mathematical expressions.

In Lean, however, programs work the same way as mathematical expressions. Once given a value, variables cannot be reassigned. Evaluating an expression cannot have side effects. If two expressions have the same value, then replacing one with the other will not cause the program to compute a different result. This does not mean that Lean cannot be used to write `Hello, world!` to the console, but performing I/O is not a core part of the experience of using Lean in the same way. Thus, this chapter focuses on how to evaluate expressions interactively with Lean, while the next chapter describes how to write, compile, and run the `Hello, world!` program.

To ask Lean to evaluate an expression, write `#eval` before it in your editor, which will then report the result back. Typically, the result is found by putting the cursor or mouse pointer over `#eval`. For instance,

```
#eval 1 + 2
```

yields the value `3`.

Lean obeys the ordinary rules of precedence and associativity for arithmetic operators. That is,

```
#eval 1 + 2 * 5
```

yields the value `11` rather than `15`.

While both ordinary mathematical notation and the majority of programming languages use parentheses (e.g. `f(x)`) to apply a function to its arguments, Lean simply writes the function next to its arguments (e.g. `f x`). Function application is one of the most common operations, so it pays to keep it concise. Rather than writing

```
#eval String.append("Hello, ", "Lean!")
```

to compute `"Hello, Lean!"`, one would instead write

```
#eval String.append "Hello, " "Lean!"
```

where the function's two arguments are simply written next to it with spaces.

Just as the order-of-operations rules for arithmetic demand parentheses in the expression `(1 + 2) * 5`, parentheses are also necessary when a function's argument is to be computed via another function call. For instance, parentheses are required in

```
#eval String.append "great " (String.append "oak " "tree")
```

because otherwise the second `String.append` would be interpreted as an argument to the first, rather than as a function being passed `"oak "` and `"tree"` as arguments. The value of the inner `String.append` call must be found first, after which it can be appended to `"great "`, yielding the final value `"great oak tree"`.

Imperative languages often have two kinds of conditional: a conditional *statement* that determines which instructions to carry out based on a Boolean value, and a conditional *expression* that determines which of two expressions to evaluate based on a Boolean value. For instance, in C and C++, the conditional statement is written using `if` and `else`, while the conditional expression is written with a ternary operator in which `?` and `:` separate the condition from the branches. In Python, the conditional statement begins with `if`, while the conditional expression puts `if` in the middle. Because Lean is an expression-oriented functional language, there are no conditional statements, only conditional expressions. They are written using `if`, `then`, and `else`. For example,

```
String.append "it is " (if 1 > 2 then "yes" else "no")
```

evaluates to

```
String.append "it is " (if false then "yes" else "no")
```

which evaluates to

```
String.append "it is " "no"
```

which finally evaluates to `"it is no"`.

For the sake of brevity, a series of evaluation steps like this will sometimes be written with arrows between them:

```
String.append "it is " (if 1 > 2 then "yes" else "no")
===>
String.append "it is " (if false then "yes" else "no")
===>
String.append "it is " "no"
===>
"it is no"
```

Messages You May Meet

Asking Lean to evaluate a function application that is missing an argument will lead to an error message. In particular, the example

```
#eval String.append "it is "
```

yields a quite long error message:

```
could not synthesize a 'ToExpr', 'Repr', or 'ToString' instance for type
  String → String
```

This message occurs because Lean functions that are applied to only some of their arguments return new functions that are waiting for the rest of the arguments. Lean cannot display functions to users, and thus returns an error when asked to do so.

Exercises

What are the values of the following expressions? Work them out by hand, then enter them into Lean to check your work.

- `42 + 19`
- `String.append "A" (String.append "B" "C")`

- `String.append (String.append "A" "B") "C"`
- `if 3 == 3 then 5 else 7`
- `if 3 == 4 then "equal" else "not equal"`

Types

Types classify programs based on the values that they can compute. Types serve a number of roles in a program:

1. They allow the compiler to make decisions about the in-memory representation of a value.

2. They help programmers to communicate their intent to others, serving as a lightweight specification for the inputs and outputs of a function. The compiler ensures that the program adheres to this specification.

3. They prevent various potential mistakes, such as adding a number to a string, and thus reduce the number of tests that are necessary for a program.

4. They help the Lean compiler automate the production of auxiliary code that can save boilerplate.

Lean's type system is unusually expressive. Types can encode strong specifications like "this sorting function returns a permutation of its input" and flexible specifications like "this function has different return types, depending on the value of its argument". The type system can even be used as a full-blown logic for proving mathematical theorems. This cutting-edge expressive power doesn't make simpler types unnecessary, however, and understanding these simpler types is a prerequisite for using the more advanced features.

Every program in Lean must have a type. In particular, every expression must have a type before it can be evaluated. In the examples so far, Lean has been able to discover a type on its own, but it is sometimes necessary to provide one. This is done using the colon operator inside parentheses:

```
#eval (1 + 2 : Nat)
```

Here, `Nat` is the type of *natural numbers*, which are arbitrary-precision unsigned integers. In Lean, `Nat` is the default type for non-negative integer literals. This default type is not always the best choice. In C, unsigned integers underflow to the largest representable numbers when subtraction would otherwise yield a result less than zero. `Nat`, however, can represent arbitrarily-large unsigned numbers, so there is no largest number to underflow to. Thus, subtraction on `Nat` returns `0` when the answer would have otherwise been negative. For instance,

```
#eval 1 - 2
```

evaluates to `0` rather than `-1`. To use a type that can represent the negative integers, provide it directly:

```
#eval (1 - 2 : Int)
```

With this type, the result is `-1`, as expected.

To check the type of an expression without evaluating it, use `#check` instead of `#eval`. For instance:

```
#check (1 - 2 : Int)
```

reports `1 - 2 : Int` without actually performing the subtraction.

When a program can't be given a type, an error is returned from both `#check` and `#eval`. For instance:

```
#check String.append ["hello", " "] "world"
```

outputs

```
application type mismatch
  String.append ["hello", " "]
argument
  ["hello", " "]
has type
  List String : Type
but is expected to have type
  String : Type
```

because the first argument to `String.append` is expected to be a string, but a list of strings was provided instead.

Functions and Definitions

In Lean, definitions are introduced using the `def` keyword. For instance, to define the name `hello` to refer to the string `"Hello"`, write:

```
def hello := "Hello"
```

In Lean, new names are defined using the colon-equal operator `:=` rather than `=`. This is because `=` is used to describe equalities between existing expressions, and using two different operators helps prevent confusion.

In the definition of `hello`, the expression `"Hello"` is simple enough that Lean is able to determine the definition's type automatically. However, most definitions are not so simple, so it will usually be necessary to add a type. This is done using a colon after the name being defined.

```
def lean : String := "Lean"
```

Now that the names have been defined, they can be used, so

```
#eval String.append hello (String.append " " lean)
```

outputs

```
"Hello Lean"
```

In Lean, defined names may only be used after their definitions.

In many languages, definitions of functions use a different syntax than definitions of other values. For instance, Python function definitions begin with the `def` keyword, while other definitions are defined with an equals sign. In Lean, functions are defined using the same `def` keyword as other values. Nonetheless, definitions such as `hello` introduce names that refer *directly* to their values, rather than to zero-argument functions that return equivalent results each time they are called.

Defining Functions

There are a variety of ways to define functions in Lean. The simplest is to place the function's arguments before the definition's type, separated by spaces. For instance, a function that adds one to its argument can be written:

```
def add1 (n : Nat) : Nat := n + 1
```

Testing this function with `#eval` gives `8`, as expected:

```
#eval add1 7
```

Just as functions are applied to multiple arguments by writing spaces between each argument, functions that accept multiple arguments are defined with spaces between the arguments' names and types. The function `maximum`, whose result is equal to the greatest of its two arguments, takes two `Nat` arguments `n` and `k` and returns a `Nat`.

```
def maximum (n : Nat) (k : Nat) : Nat :=
  if n < k then
    k
  else n
```

Similarly, the function `spaceBetween` joins two strings with a space between them.

```
def spaceBetween (before : String) (after : String) : String :=
  String.append before (String.append " " after)
```

When a defined function like `maximum` has been provided with its arguments, the result is determined by first replacing the argument names with the provided values in the body, and then evaluating the resulting body. For example:

```
maximum (5 + 8) (2 * 7)
===>
maximum 13 14
===>
if 13 < 14 then 14 else 13
===>
14
```

Expressions that evaluate to natural numbers, integers, and strings have types that say this (`Nat`, `Int`, and `String`, respectively). This is also true of functions. A function that accepts a `Nat` and returns a `Bool` has type `Nat → Bool`, and a function that accepts two `Nat`s and returns a `Nat` has type `Nat → Nat → Nat`.

As a special case, Lean returns a function's signature when its name is used directly with `#check`. Entering `#check add1` yields `add1 (n : Nat) : Nat`. However, Lean can be "tricked" into showing the function's type by writing the function's name in parentheses, which causes the function to be treated as an ordinary expression, so `#check (add1)` yields `add1 : Nat → Nat` and `#check (maximum)` yields `maximum : Nat → Nat → Nat`. This arrow can also be written with an ASCII alternative arrow `->`, so the preceding function types can be written `Nat -> Nat` and `Nat -> Nat -> Nat`, respectively.

Behind the scenes, all functions actually expect precisely one argument. Functions like `maximum` that seem to take more than one argument are in fact functions that take one argument and then return a new function. This new function takes the next argument, and the process continues until no more arguments are expected. This can be seen by providing one argument to a multiple-argument function: `#check maximum 3` yields `maximum 3 : Nat → Nat` and `#check spaceBetween "Hello "` yields `spaceBetween "Hello " : String → String`. Using a function that returns a function to implement multiple-argument functions is called *currying* after the mathematician Haskell Curry. Function arrows associate to the right, which means that `Nat → Nat → Nat` should be parenthesized `Nat → (Nat → Nat)`.

Exercises

- Define the function `joinStringsWith` with type `String -> String -> String -> String` that creates a new string by placing its first argument between its second and third arguments. `joinStringsWith ", " "one" "and another"` should evaluate to `"one, and another"`.
- What is the type of `joinStringsWith ": "`? Check your answer with Lean.
- Define a function `volume` with type `Nat → Nat → Nat → Nat` that computes the volume of a rectangular prism with the given height, width, and depth.

Defining Types

Most typed programming languages have some means of defining aliases for types, such as C's `typedef`. In Lean, however, types are a first-class part of the language - they are expressions like any other. This means that definitions can refer to types just as well as they can refer to other values.

For example, if `String` is too much to type, a shorter abbreviation `Str` can be defined:

```
def Str : Type := String
```

It is then possible to use `Str` as a definition's type instead of `String`:

```
def aStr : Str := "This is a string."
```

The reason this works is that types follow the same rules as the rest of Lean. Types are expressions, and in an expression, a defined name can be replaced with its definition. Because `Str` has been defined to mean `String`, the definition of `aStr` makes sense.

Messages You May Meet

Experimenting with using definitions for types is made more complicated by the way that Lean supports overloaded integer literals. If `Nat` is too short, a longer name `NaturalNumber` can be defined:

```
def NaturalNumber : Type := Nat
```

However, using `NaturalNumber` as a definition's type instead of `Nat` does not have the expected effect. In particular, the definition:

```
def thirtyEight : NaturalNumber := 38
```

results in the following error:

```
failed to synthesize
  OfNat NaturalNumber 38
numerals are polymorphic in Lean, but the numeral `38` cannot be used in a
context where the expected type is
  NaturalNumber
due to the absence of the instance above

Additional diagnostic information may be available using the `set_option
diagnostics true` command.
```

This error occurs because Lean allows number literals to be *overloaded*. When it makes sense to do so, natural number literals can be used for new types, just as if those types were built in to the system. This is part of Lean's mission of making it convenient to represent mathematics, and different branches of mathematics use number notation for very different purposes. The specific feature that allows this overloading does not replace all defined names with their definitions before looking for overloading, which is what leads to the error message above.

One way to work around this limitation is by providing the type `Nat` on the right-hand side of the definition, causing `Nat`'s overloading rules to be used for `38`:

```
def thirtyEight : NaturalNumber := (38 : Nat)
```

The definition is still type-correct because `NaturalNumber` is the same type as `Nat`—by definition!

Another solution is to define an overloading for `NaturalNumber` that works equivalently to the one for `Nat` . This requires more advanced features of Lean, however.

Finally, defining the new name for `Nat` using `abbrev` instead of `def` allows overloading resolution to replace the defined name with its definition. Definitions written using `abbrev` are always unfolded. For instance,

```
abbrev N : Type := Nat
```

and

```
def thirtyNine : N := 39
```

are accepted without issue.

Behind the scenes, some definitions are internally marked as being unfoldable during overload resolution, while others are not. Definitions that are to be unfolded are called *reducible*. Control over reducibility is essential to allow Lean to scale: fully unfolding all definitions can result in very large types that are slow for a machine to process and difficult for users to understand. Definitions produced with `abbrev` are marked as reducible.

Structures

The first step in writing a program is usually to identify the problem domain's concepts, and then find suitable representations for them in code. Sometimes, a domain concept is a collection of other, simpler, concepts. In that case, it can be convenient to group these simpler components together into a single "package", which can then be given a meaningful name. In Lean, this is done using *structures*, which are analogous to `struct` s in C or Rust and `record` s in C#.

Defining a structure introduces a completely new type to Lean that can't be reduced to any other type. This is useful because multiple structures might represent different concepts that nonetheless contain the same data. For instance, a point might be represented using either Cartesian or polar coordinates, each being a pair of floating-point numbers. Defining separate structures prevents API clients from confusing one for another.

Lean's floating-point number type is called `Float` , and floating-point numbers are written in the usual notation.

```
#check 1.2
```

```
1.2 : Float
```

```
#check -454.2123215
```

```
-454.2123215 : Float
```

```
#check 0.0
```

```
0.0 : Float
```

When floating point numbers are written with the decimal point, Lean will infer the type `Float` . If they are written without it, then a type annotation may be necessary.

```
#check 0
```

```
0 : Nat
```

```
#check (0 : Float)
```

```
0 : Float
```

A Cartesian point is a structure with two `Float` fields, called `x` and `y`. This is declared using the `structure` keyword.

```
structure Point where
  x : Float
  y : Float
deriving Repr
```

After this declaration, `Point` is a new structure type. The final line, which says `deriving Repr`, asks Lean to generate code to display values of type `Point`. This code is used by `#eval` to render the result of evaluation for consumption by programmers, analogous to the `repr` function in Python. It is also possible to override the compiler's generated display code.

The typical way to create a value of a structure type is to provide values for all of its fields inside of curly braces. The origin of a Cartesian plane is where `x` and `y` are both zero:

```
def origin : Point := { x := 0.0, y := 0.0 }
```

If the `deriving Repr` line in `Point`'s definition were omitted, then attempting `--    #eval origin` would yield an error similar to that which occurs when omitting a function's argument:

```
- "expression
--   origin
-- has type
--   Point
-- but instance
--   Lean.MetaEval Point
-- failed to be synthesized, this instance instructs Lean on how to display the
resulting value, recall that any type implementing the `Repr` class also
implements the `Lean.MetaEval` class
```

That message is saying that the evaluation machinery doesn't know how to communicate the result of evaluation back to the user.

Happily, with `deriving Repr`, the result of `#eval origin` looks very much like the definition of `origin`.

```
{ x := 0.000000, y := 0.000000 }
```

Because structures exist to "bundle up" a collection of data, naming it and treating it as a single unit, it is also important to be able to extract the individual fields of a structure. This is done using dot notation, as in C, Python, Rust, or JavaScript.

```
#eval origin.x
```

```
0.000000
```

```
#eval origin.y
```

```
0.000000
```

This can be used to define functions that take structures as arguments. For instance, addition of points is performed by adding the underlying coordinate values. It should be the case that `#eval addPoints { x := 1.5, y := 32 } { x := -8, y := 0.2 }` yields

```
{ x := -6.500000, y := 32.200000 }
```

The function itself takes two `Points` as arguments, called `p1` and `p2`. The resulting point is based on the `x` and `y` fields of both `p1` and `p2`:

```
def addPoints (p1 : Point) (p2 : Point) : Point :=
  { x := p1.x + p2.x, y := p1.y + p2.y }
```

Similarly, the distance between two points, which is the square root of the sum of the squares of the differences in their `x` and `y` components, can be written:

```
def distance (p1 : Point) (p2 : Point) : Float :=
  Float.sqrt (((p2.x - p1.x) ^ 2.0) + ((p2.y - p1.y) ^ 2.0))
```

For example, the distance between (1, 2) and (5, -1) is 5:

```
#eval distance { x := 1.0, y := 2.0 } { x := 5.0, y := -1.0 }
```

```
5.000000
```

Multiple structures may have fields with the same names. A three-dimensional point datatype may share the fields `x` and `y`, and be instantiated with the same field names:

```
structure Point3D where
  x : Float
  y : Float
  z : Float
deriving Repr

def origin3D : Point3D := { x := 0.0, y := 0.0, z := 0.0 }
```

This means that the structure's expected type must be known in order to use the curly-brace syntax. If the type is not known, Lean will not be able to instantiate the structure. For example,

```
#check { x := 0.0, y := 0.0 }
```

leads to the error

```
invalid {...} notation, expected type is not known
```

As usual, the situation can be remedied by providing a type annotation.

```
#check ({ x := 0.0, y := 0.0 } : Point)
```

```
{ x := 0.0, y := 0.0 } : Point
```

To make programs more concise, Lean also allows the structure type annotation inside the curly braces.

```
#check { x := 0.0, y := 0.0 : Point}
```

```
{ x := 0.0, y := 0.0 } : Point
```

Updating Structures

Imagine a function `zeroX` that replaces the `x` field of a `Point` with `0.0`. In most programming language communities, this sentence would mean that the memory location pointed to by `x` was to be overwritten with a new value. However, Lean is a functional programming language. In functional programming communities, what is almost always meant by this kind of statement is that a fresh `Point` is allocated with the `x` field pointing to the new value, and all other fields pointing to the original values from the input. One way to write `zeroX` is to follow this description literally, filling out the new value for `x` and manually transferring `y`:

```
def zeroX (p : Point) : Point :=
  { x := 0, y := p.y }
```

This style of programming has drawbacks, however. First off, if a new field is added to a structure, then every site that updates any field at all must be updated, causing maintenance difficulties. Secondly, if the structure contains multiple fields with the same type, then there is a real risk of copy-paste coding leading to field contents being duplicated or switched. Finally, the program becomes long and bureaucratic.

Lean provides a convenient syntax for replacing some fields in a structure while leaving the others alone. This is done by using the `with` keyword in a structure initialization. The source of unchanged fields occurs before the `with`, and the new fields occur after. For example, `zeroX` can be written with only the new `x` value:

```
def zeroX (p : Point) : Point :=
  { p with x := 0 }
```

Remember that this structure update syntax does not modify existing values—it creates new values that share some fields with old values. Given the point `fourAndThree`:

```
def fourAndThree : Point :=
  { x := 4.3, y := 3.4 }
```

evaluating it, then evaluating an update of it using `zeroX`, then evaluating it again yields the original value:

```
#eval fourAndThree
```

```
{ x := 4.300000, y := 3.400000 }
```

```
#eval zeroX fourAndThree
```

```
{ x := 0.000000, y := 3.400000 }
```

```
#eval fourAndThree
```

```
{ x := 4.300000, y := 3.400000 }
```

One consequence of the fact that structure updates do not modify the original structure is that it becomes easier to reason about cases where the new value is computed from the old one. All references to the old structure continue to refer to the same field values in all of the new values provided.

Behind the Scenes

Every structure has a *constructor*. Here, the term "constructor" may be a source of confusion.

Unlike constructors in languages such as Java or Python, constructors in Lean are not arbitrary code to be run when a datatype is initialized. Instead, constructors simply gather the data to be stored in the newly-allocated data structure. It is not possible to provide a custom constructor that pre-processes data or rejects invalid arguments. This is really a case of the word "constructor" having different, but related, meanings in the two contexts.

By default, the constructor for a structure named `S` is named `S.mk`. Here, `S` is a namespace qualifier, and `mk` is the name of the constructor itself. Instead of using curly-brace initialization syntax, the constructor can also be applied directly.

```
#check Point.mk 1.5 2.8
```

However, this is not generally considered to be good Lean style, and Lean even returns its feedback using the standard structure initializer syntax.

```
{ x := 1.5, y := 2.8 } : Point
```

Constructors have function types, which means they can be used anywhere that a function is expected. For instance, `Point.mk` is a function that accepts two `Float`s (respectively `x` and `y`) and returns a new `Point`.

```
#check (Point.mk)
```

```
Point.mk : Float → Float → Point
```

To override a structure's constructor name, write it with two colons at the beginning. For instance, to use `Point.point` instead of `Point.mk`, write:

```
structure Point where
  point ::
  x : Float
  y : Float
deriving Repr
```

In addition to the constructor, an accessor function is defined for each field of a structure. These have the same name as the field, in the structure's namespace. For `Point`, accessor functions `Point.x` and `Point.y` are generated.

```
#check (Point.x)
```

```
Point.x : Point → Float
```

```
#check (Point.y)
```

```
Point.y : Point → Float
```

In fact, just as the curly-braced structure construction syntax is converted to a call to the structure's constructor behind the scenes, the syntax `p1.x` in the prior definition of `addPoints` is converted into a call to the `Point.x` accessor. That is, `#eval origin.x` and `#eval Point.x origin` both yield

```
0.000000
```

Accessor dot notation is usable with more than just structure fields. It can also be used for functions that take any number of arguments. More generally, accessor notation has the form `TARGET.f ARG1 ARG2 ...`. If `TARGET` has type `T`, the function named `T.f` is called. `TARGET` becomes its leftmost argument of type `T`, which is often but not always the first one, and `ARG1 ARG2 ...` are provided in order as the remaining arguments. For instance, `String.append` can be invoked from a string with accessor notation, even though `String` is not a structure with an `append` field.

```
#eval "one string".append " and another"
```

```
"one string and another"
```

In that example, `TARGET` represents `"one string"` and `ARG1` represents `" and another"`.

The function `Point.modifyBoth` (that is, `modifyBoth` defined in the `Point` namespace) applies a function to both fields in a `Point`:

```
def Point.modifyBoth (f : Float → Float) (p : Point) : Point :=
  { x := f p.x, y := f p.y }
```

Even though the `Point` argument comes after the function argument, it can be used with dot notation as well:

```
#eval fourAndThree.modifyBoth Float.floor
```

```
{ x := 4.000000, y := 3.000000 }
```

In this case, `TARGET` represents `fourAndThree`, while `ARG1` is `Float.floor`. This is because the target of the accessor notation is used as the first argument in which the type matches, not necessarily the first argument.

Exercises

- Define a structure named `RectangularPrism` that contains the height, width, and depth of a rectangular prism, each as a `Float`.
- Define a function named `volume : RectangularPrism → Float` that computes the volume of a rectangular prism.
- Define a structure named `Segment` that represents a line segment by its endpoints, and define a function `length : Segment → Float` that computes the length of a line segment. `Segment` should have at most two fields.
- Which names are introduced by the declaration of `RectangularPrism`?
- Which names are introduced by the following declarations of `Hamster` and `Book`? What are their types?

```
structure Hamster where
  name : String
  fluffy : Bool
```

```
structure Book where
  makeBook ::
  title : String
  author : String
  price : Float
```

Datatypes and Patterns

Structures enable multiple independent pieces of data to be combined into a coherent whole that is represented by a brand new type. Types such as structures that group together a collection of values are called *product types*. Many domain concepts, however, can't be naturally represented as structures. For instance, an application might need to track user permissions, where some users are document owners, some may edit documents, and others may only read them. A calculator has a number of binary operators, such as addition, subtraction, and multiplication. Structures do not provide an easy way to encode multiple choices.

Similarly, while a structure is an excellent way to keep track of a fixed set of fields, many applications require data that may contain an arbitrary number of elements. Most classic data structures, such as trees and lists, have a recursive structure, where the tail of a list is itself a list, or where the left and right branches of a binary tree are themselves binary trees. In the aforementioned calculator, the structure of expressions themselves is recursive. The summands in an addition expression may themselves be multiplication expressions, for instance.

Datatypes that allow choices are called *sum types* and datatypes that can include instances of themselves are called *recursive datatypes*. Recursive sum types are called *inductive datatypes*, because mathematical induction may be used to prove statements about them. When programming, inductive datatypes are consumed through pattern matching and recursive functions.

Many of the built-in types are actually inductive datatypes in the standard library. For instance, `Bool` is an inductive datatype:

```
inductive Bool where
  | false : Bool
  | true : Bool
```

This definition has two main parts. The first line provides the name of the new type (`Bool`), while the remaining lines each describe a constructor. As with constructors of structures, constructors of inductive datatypes are mere inert receivers of and containers for other data, rather than places to insert arbitrary initialization and validation code. Unlike structures, inductive datatypes may have multiple constructors. Here, there are two constructors, `true` and `false` , and neither takes any arguments. Just as a structure declaration places its names in a namespace named after the declared type, an inductive datatype places the names of its constructors in a namespace. In the Lean standard library, `true` and `false` are re-exported from this namespace so that they can be written alone, rather than as `Bool.true` and `Bool.false`, respectively.

From a data modeling perspective, inductive datatypes are used in many of the same contexts where a sealed abstract class might be used in other languages. In languages like C# or Java, one might write a similar definition of `Bool`:

```
abstract class Bool {}
class True : Bool {}
class False : Bool {}
```

However, the specifics of these representations are fairly different. In particular, each non-abstract class creates both a new type and new ways of allocating data. In the object-oriented example, `True` and `False` are both types that are more specific than `Bool`, while the Lean definition introduces only the new type `Bool`.

The type `Nat` of non-negative integers is an inductive datatype:

```
inductive Nat where
  | zero : Nat
  | succ (n : Nat) : Nat
```

Here, `zero` represents 0, while `succ` represents the successor of some other number. The `Nat` mentioned in `succ`'s declaration is the very type `Nat` that is in the process of being defined. *Successor* means "one greater than", so the successor of five is six and the successor of 32,185 is 32,186. Using this definition, `4` is represented as `Nat.succ (Nat.succ (Nat.succ (Nat.succ Nat.zero)))`. This definition is almost like the definition of `Bool` with slightly different names. The only real difference is that `succ` is followed by `(n : Nat)`, which specifies that the constructor `succ` takes an argument of type `Nat` which happens to be named `n`. The names `zero` and `succ` are in a namespace named after their type, so they must be referred to as `Nat.zero` and `Nat.succ`, respectively.

Argument names, such as `n`, may occur in Lean's error messages and in feedback provided when writing mathematical proofs. Lean also has an optional syntax for providing arguments by name. Generally, however, the choice of argument name is less important than the choice of a structure field name, as it does not form as large a part of the API.

In C# or Java, `Nat` could be defined as follows:

```
abstract class Nat {}
class Zero : Nat {}
class Succ : Nat {
  public Nat n;
  public Succ(Nat pred) {
    n = pred;
  }
}
```

Just as in the `Bool` example above, this defines more types than the Lean equivalent. Additionally, this example highlights how Lean datatype constructors are much more like subclasses of an abstract class than they are like constructors in C# or Java, as the constructor shown here contains initialization code to be executed.

Sum types are also similar to using a string tag to encode discriminated unions in TypeScript. In TypeScript, `Nat` could be defined as follows:

```
interface Zero {
    tag: "zero";
}

interface Succ {
    tag: "succ";
    predecessor: Nat;
}

type Nat = Zero | Succ;
```

Just like C# and Java, this encoding ends up with more types than in Lean, because `Zero` and `Succ` are each a type on their own. It also illustrates that Lean constructors correspond to objects in JavaScript or TypeScript that include a tag that identifies the contents.

Pattern Matching

In many languages, these kinds of data are consumed by first using an instance-of operator to check which subclass has been received and then reading the values of the fields that are available in the given subclass. The instance-of check determines which code to run, ensuring that the data needed by this code is available, while the fields themselves provide the data. In Lean, both of these purposes are simultaneously served by *pattern matching*.

An example of a function that uses pattern matching is `isZero`, which is a function that returns `true` when its argument is `Nat.zero`, or false otherwise.

```
def isZero (n : Nat) : Bool :=
  match n with
  | Nat.zero => true
  | Nat.succ k => false
```

The `match` expression is provided the function's argument `n` for destructuring. If `n` was constructed by `Nat.zero`, then the first branch of the pattern match is taken, and the result is `true`. If `n` was constructed by `Nat.succ`, then the second branch is taken, and the result is `false`.

Step-by-step, evaluation of `isZero Nat.zero` proceeds as follows:

```
isZero Nat.zero
===>
match Nat.zero with
| Nat.zero => true
| Nat.succ k => false
===>
true
```

Evaluation of `isZero 5` proceeds similarly:

```
isZero 5
===>
isZero (Nat.succ (Nat.succ (Nat.succ (Nat.succ (Nat.succ Nat.zero)))))
===>
match Nat.succ (Nat.succ (Nat.succ (Nat.succ (Nat.succ Nat.zero)))) with
| Nat.zero => true
| Nat.succ k => false
===>
false
```

The `k` in the second branch of the pattern in `isZero` is not decorative. It makes the `Nat` that is the argument to `succ` visible, with the provided name. That smaller number can then be used to compute the final result of the expression.

Just as the successor of some number n is one greater than n (that is, $n + 1$), the predecessor of a number is one less than it. If `pred` is a function that finds the predecessor of a `Nat`, then it should be the case that the following examples find the expected result:

```
#eval pred 5
```

```
4
```

```
#eval pred 839
```

```
838
```

Because `Nat` cannot represent negative numbers, `0` is a bit of a conundrum. Usually, when working with `Nat`, operators that would ordinarily produce a negative number are redefined to produce `0` itself:

```
#eval pred 0
```

```
0
```

To find the predecessor of a `Nat`, the first step is to check which constructor was used to create it. If it was `Nat.zero`, then the result is `Nat.zero`. If it was `Nat.succ`, then the name `k` is used to refer to the `Nat` underneath it. And this `Nat` is the desired predecessor, so the result of the `Nat.succ` branch is `k`.

```
def pred (n : Nat) : Nat :=
  match n with
  | Nat.zero => Nat.zero
  | Nat.succ k => k
```

Applying this function to `5` yields the following steps:

```
pred 5
===>
pred (Nat.succ 4)
===>
match Nat.succ 4 with
| Nat.zero => Nat.zero
| Nat.succ k => k
===>
4
```

Pattern matching can be used with structures as well as with sum types. For instance, a function that extracts the third dimension from a `Point3D` can be written as follows:

```
def depth (p : Point3D) : Float :=
  match p with
  | { x:= h, y := w, z := d } => d
```

In this case, it would have been much simpler to just use the `z` accessor, but structure patterns are occasionally the simplest way to write a function.

Recursive Functions

Definitions that refer to the name being defined are called *recursive definitions*. Inductive datatypes are allowed to be recursive; indeed, `Nat` is an example of such a datatype because `succ` demands another `Nat`. Recursive datatypes can represent arbitrarily large data, limited only by technical factors like available memory. Just as it would be impossible to write down one constructor for each natural number in the datatype definition, it is also impossible to write down a pattern match case for each possibility.

Recursive datatypes are nicely complemented by recursive functions. A simple recursive function over `Nat` checks whether its argument is even. In this case, `zero` is even. Non-recursive branches of the code like this one are called *base cases*. The successor of an odd number is even, and the successor of an even number is odd. This means that a number built with `succ` is even if and only if its argument is not even.

```
def even (n : Nat) : Bool :=
  match n with
  | Nat.zero => true
  | Nat.succ k => not (even k)
```

This pattern of thought is typical for writing recursive functions on `Nat`. First, identify what to do for `zero`. Then, determine how to transform a result for an arbitrary `Nat` into a result for its successor, and apply this transformation to the result of the recursive call. This pattern is called *structural recursion*.

Unlike many languages, Lean ensures by default that every recursive function will eventually reach a base case. From a programming perspective, this rules out accidental infinite loops. But this feature is especially important when proving theorems, where infinite loops cause major difficulties. A consequence of this is that Lean will not accept a version of `even` that attempts to invoke itself recursively on the original number:

```
def evenLoops (n : Nat) : Bool :=
  match n with
  | Nat.zero => true
  | Nat.succ k => not (evenLoops n)
```

The important part of the error message is that Lean could not determine that the recursive function always reaches a base case (because it doesn't).

```
fail to show termination for
  evenLoops
with errors
failed to infer structural recursion:
Not considering parameter n of evenLoops:
  it is unchanged in the recursive calls
no parameters suitable for structural recursion

well-founded recursion cannot be used, 'evenLoops' does not take any (non-fixed)
arguments
```

Even though addition takes two arguments, only one of them needs to be inspected. To add zero to a number n, just return n. To add the successor of k to n, take the successor of the result of adding k to n.

```
def plus (n : Nat) (k : Nat) : Nat :=
  match k with
  | Nat.zero => n
  | Nat.succ k' => Nat.succ (plus n k')
```

In the definition of `plus`, the name `k'` is chosen to indicate that it is connected to, but not identical with, the argument `k`. For instance, walking through the evaluation of `plus 3 2` yields the following steps:

```
plus 3 2
===>
plus 3 (Nat.succ (Nat.succ Nat.zero))
===>
match Nat.succ (Nat.succ Nat.zero) with
| Nat.zero => 3
| Nat.succ k' => Nat.succ (plus 3 k')
===>
Nat.succ (plus 3 (Nat.succ Nat.zero))
===>
Nat.succ (match Nat.succ Nat.zero with
| Nat.zero => 3
| Nat.succ k' => Nat.succ (plus 3 k'))
===>
Nat.succ (Nat.succ (plus 3 Nat.zero))
===>
Nat.succ (Nat.succ (match Nat.zero with
| Nat.zero => 3
| Nat.succ k' => Nat.succ (plus 3 k')))
===>
Nat.succ (Nat.succ 3)
===>
5
```

One way to think about addition is that $n + k$ applies `Nat.succ` k times to n. Similarly, multiplication $n \times k$ adds n to itself k times and subtraction $n - k$ takes n's predecessor k times.

```
def times (n : Nat) (k : Nat) : Nat :=
  match k with
  | Nat.zero => Nat.zero
  | Nat.succ k' => plus n (times n k')

def minus (n : Nat) (k : Nat) : Nat :=
  match k with
  | Nat.zero => n
  | Nat.succ k' => pred (minus n k')
```

Not every function can be easily written using structural recursion. The understanding of addition as iterated `Nat.succ`, multiplication as iterated addition, and subtraction as iterated predecessor suggests an implementation of division as iterated subtraction. In this case, if the numerator is less than the divisor, the result is zero. Otherwise, the result is the successor of dividing the numerator minus the divisor by the divisor.

```
def div (n : Nat) (k : Nat) : Nat :=
  if n < k then
    0
  else Nat.succ (div (n - k) k)
```

As long as the second argument is not `0`, this program terminates, as it always makes progress towards the base case. However, it is not structurally recursive, because it doesn't follow the pattern of finding a result for zero and transforming a result for a smaller `Nat` into a result for its successor. In particular, the recursive invocation of the function is applied to the result of another function call, rather than to an input constructor's argument. Thus, Lean rejects it with the following message:

```
fail to show termination for
  div
with errors
failed to infer structural recursion:
Cannot use parameter n:
  failed to eliminate recursive application
    div (n - k) k
Cannot use parameter k:
  failed to eliminate recursive application
    div (n - k) k


Could not find a decreasing measure.
The basic measures relate at each recursive call as follows:
(<, ≤, =: relation proved, ? all proofs failed, _: no proof attempted)
             n k
1) 720:19-32 ≤ =
Please use `termination_by` to specify a decreasing measure.
```

This message means that `div` requires a manual proof of termination. This topic is explored in the final chapter.

Polymorphism

Just as in most languages, types in Lean can take arguments. For instance, the type `List Nat` describes lists of natural numbers, `List String` describes lists of strings, and `List (List Point)` describes lists of lists of points. This is very similar to `List<Nat>`, `List<String>`, or `List<List<Point>>` in a language like C# or Java. Just as Lean uses a space to pass an argument to a function, it uses a space to pass an argument to a type.

In functional programming, the term *polymorphism* typically refers to datatypes and definitions that take types as arguments. This is different from the object-oriented programming community, where the term typically refers to subclasses that may override some behavior of their superclass. In this book, "polymorphism" always refers to the first sense of the word. These type arguments can be used in the datatype or definition, which allows the same datatype or definition to be used with any type that results from replacing the arguments' names with some other types.

The `Point` structure requires that both the `x` and `y` fields are `Float`s. There is, however, nothing about points that require a specific representation for each coordinate. A polymorphic version of `Point`, called `PPoint`, can take a type as an argument, and then use that type for both fields:

```
structure PPoint (α : Type) where
  x : α
  y : α
deriving Repr
```

Just as a function definition's arguments are written immediately after the name being defined, a structure's arguments are written immediately after the structure's name. It is customary to use Greek letters to name type arguments in Lean when no more specific name suggests itself. `Type` is a type that describes other types, so `Nat`, `List String`, and `PPoint Int` all have type `Type`.

Just like `List`, `PPoint` can be used by providing a specific type as its argument:

```
def natOrigin : PPoint Nat :=
  { x := Nat.zero, y := Nat.zero }
```

In this example, both fields are expected to be `Nat`s. Just as a function is called by replacing its argument variables with its argument values, providing `PPoint` with the type `Nat` as an argument yields a structure in which the fields `x` and `y` have the type `Nat`, because the argument name `α` has been replaced by the argument type `Nat`. Types are ordinary expressions in Lean, so passing arguments to polymorphic types (like `PPoint`) doesn't require any special syntax.

Definitions may also take types as arguments, which makes them polymorphic. The function `replaceX` replaces the `x` field of a `PPoint` with a new value. In order to allow `replaceX` to work with *any* polymorphic point, it must be polymorphic itself. This is achieved by having its first argument be the type of the point's fields, with later arguments referring back to the first argument's name.

```
def replaceX (α : Type) (point : PPoint α) (newX : α) : PPoint α :=
  { point with x := newX }
```

In other words, when the types of the arguments `point` and `newX` mention `α`, they are referring to *whichever type was provided as the first argument*. This is similar to the way that function argument names refer to the values that were provided when they occur in the function's body.

This can be seen by asking Lean to check the type of `replaceX`, and then asking it to check the type of `replaceX Nat`.

```
#check (replaceX)
```

```
replaceX : (α : Type) → PPoint α → α → PPoint α
```

This function type includes the *name* of the first argument, and later arguments in the type refer back to this name. Just as the value of a function application is found by replacing the argument name with the provided argument value in the function's body, the type of a function application is found by replacing the argument's name with the provided value in the function's return type. Providing the first argument, `Nat`, causes all occurrences of `α` in the remainder of the type to be replaced with `Nat`:

```
#check replaceX Nat
```

```
replaceX Nat : PPoint Nat → Nat → PPoint Nat
```

Because the remaining arguments are not explicitly named, no further substitution occurs as more arguments are provided:

```
#check replaceX Nat natOrigin
```

```
replaceX Nat natOrigin : Nat → PPoint Nat
```

```
#check replaceX Nat natOrigin 5
```

```
replaceX Nat natOrigin 5 : PPoint Nat
```

The fact that the type of the whole function application expression was determined by passing a type as an argument has no bearing on the ability to evaluate it.

```
#eval replaceX Nat natOrigin 5
```

```
{ x := 5, y := 0 }
```

Polymorphic functions work by taking a named type argument and having later types refer to the argument's name. However, there's nothing special about type arguments that allows them to be named. Given a datatype that represents positive or negative signs:

```
inductive Sign where
  | pos
  | neg
```

it is possible to write a function whose argument is a sign. If the argument is positive, the function returns a `Nat`, while if it's negative, it returns an `Int`:

```
def posOrNegThree (s : Sign) : match s with | Sign.pos => Nat | Sign.neg => Int
:=
  match s with
  | Sign.pos => (3 : Nat)
  | Sign.neg => (-3 : Int)
```

Because types are first class and can be computed using the ordinary rules of the Lean language, they can be computed by pattern-matching against a datatype. When Lean is checking this function, it uses the fact that the `match`-expression in the function's body corresponds to the `match`-expression in the type to make `Nat` be the expected type for the `pos` case and to make `Int` be the expected type for the `neg` case.

Applying `posOrNegThree` to `Sign.pos` results in the argument name `s` in both the body of the function and its return type being replaced by `Sign.pos`. Evaluation can occur both in the expression and its type:

```
(posOrNegThree Sign.pos : match Sign.pos with | Sign.pos => Nat | Sign.neg =>
Int)
===>
((match Sign.pos with
  | Sign.pos => (3 : Nat)
  | Sign.neg => (-3 : Int)) :
 match Sign.pos with | Sign.pos => Nat | Sign.neg => Int)
===>
((3 : Nat) : Nat)
===>
3
```

Linked Lists

Lean's standard library includes a canonical linked list datatype, called `List`, and special syntax that makes it more convenient to use. Lists are written in square brackets. For instance, a list that contains the prime numbers less than 10 can be written:

```
def primesUnder10 : List Nat := [2, 3, 5, 7]
```

Behind the scenes, `List` is an inductive datatype, defined like this:

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

The actual definition in the standard library is slightly different, because it uses features that have not yet been presented, but it is substantially similar. This definition says that `List` takes a single type as its argument, just as `PPoint` did. This type is the type of the entries stored in the list. According to the constructors, a `List α` can be built with either `nil` or `cons`. The constructor `nil` represents empty lists and the constructor `cons` is used for non-empty lists. The first argument to `cons` is the head of the list, and the second argument is its tail. A list that contains n entries contains n `cons` constructors, the last of which has `nil` as its tail.

The `primesUnder10` example can be written more explicitly by using `List`'s constructors directly:

```
def explicitPrimesUnder10 : List Nat :=
  List.cons 2 (List.cons 3 (List.cons 5 (List.cons 7 List.nil)))
```

These two definitions are completely equivalent, but `primesUnder10` is much easier to read than `explicitPrimesUnder10`.

Functions that consume `List`s can be defined in much the same way as functions that consume `Nat`s. Indeed, one way to think of a linked list is as a `Nat` that has an extra data field dangling off each `succ` constructor. From this point of view, computing the length of a list is the process of replacing each `cons` with a `succ` and the final `nil` with a `zero`. Just as `replaceX` took the type of the fields of the point as an argument, `length` takes the type of the list's entries. For example, if the list contains strings, then the first argument is `String`: `length String ["Sourdough", "bread"]`. It should compute like this:

```
length String ["Sourdough", "bread"]
===>
length String (List.cons "Sourdough" (List.cons "bread" List.nil))
===>
Nat.succ (length String (List.cons "bread" List.nil))
===>
Nat.succ (Nat.succ (length String List.nil))
===>
Nat.succ (Nat.succ Nat.zero)
===>
2
```

The definition of `length` is both polymorphic (because it takes the list entry type as an argument) and recursive (because it refers to itself). Generally, functions follow the shape of the data: recursive datatypes lead to recursive functions, and polymorphic datatypes lead to polymorphic functions.

```
def length (α : Type) (xs : List α) : Nat :=
  match xs with
  | List.nil => Nat.zero
  | List.cons y ys => Nat.succ (length α ys)
```

Names such as `xs` and `ys` are conventionally used to stand for lists of unknown values. The `s` in the name indicates that they are plural, so they are pronounced "exes" and "whys" rather than "x s" and "y s".

To make it easier to read functions on lists, the bracket notation `[]` can be used to pattern-match against `nil`, and an infix `::` can be used in place of `cons`:

```
def length (α : Type) (xs : List α) : Nat :=
  match xs with
  | [] => 0
  | y :: ys => Nat.succ (length α ys)
```

Implicit Arguments

Both `replaceX` and `length` are somewhat bureaucratic to use, because the type argument is typically uniquely determined by the later values. Indeed, in most languages, the compiler is perfectly capable of determining type arguments on its own, and only occasionally needs help from users. This is also the case in Lean. Arguments can be declared *implicit* by wrapping them in curly braces instead of parentheses when defining a function. For example, a version of `replaceX` with an implicit type argument looks like this:

```
def replaceX {α : Type} (point : PPoint α) (newX : α) : PPoint α :=
  { point with x := newX }
```

It can be used with `natOrigin` without providing `Nat` explicitly, because Lean can *infer* the value of `α` from the later arguments:

```
#eval replaceX natOrigin 5
```

```
{ x := 5, y := 0 }
```

Similarly, `length` can be redefined to take the entry type implicitly:

```
def length {α : Type} (xs : List α) : Nat :=
  match xs with
  | [] => 0
  | y :: ys => Nat.succ (length ys)
```

This `length` function can be applied directly to `primesUnder10`:

```
#eval length primesUnder10
```

```
4
```

In the standard library, Lean calls this function `List.length`, which means that the dot syntax that is used for structure field access can also be used to find the length of a list:

```
#eval primesUnder10.length
```

```
4
```

Just as C# and Java require type arguments to be provided explicitly from time to time, Lean is not always capable of finding implicit arguments. In these cases, they can be provided using their names. For example, a version of `List.length` that only works for lists of integers can be specified by setting `α` to `Int`:

```
#check List.length (α := Int)
```

```
List.length : List Int → Nat
```

More Built-In Datatypes

In addition to lists, Lean's standard library contains a number of other structures and inductive datatypes that can be used in a variety of contexts.

`Option`

Not every list has a first entry—some lists are empty. Many operations on collections may fail to find what they are looking for. For instance, a function that finds the first entry in a list may not find any such entry. It must therefore have a way to signal that there was no first entry.

Many languages have a `null` value that represents the absence of a value. Instead of equipping existing types with a special `null` value, Lean provides a datatype called `Option` that equips some other type with an indicator for missing values. For instance, a nullable `Int` is represented by `Option Int`, and a nullable list of strings is represented by the type `Option (List String)`. Introducing a new type to represent nullability means that the type system ensures that checks for `null` cannot be forgotten, because an `Option Int` can't be used in a context where an `Int` is expected.

`Option` has two constructors, called `some` and `none`, that respectively represent the non-null and null versions of the underlying type. The non-null constructor, `some`, contains the underlying value, while `none` takes no arguments:

```
inductive Option (α : Type) : Type where
  | none : Option α
  | some (val : α) : Option α
```

The `Option` type is very similar to nullable types in languages like C# and Kotlin, but it is not identical. In these languages, if a type (say, `Boolean`) always refers to actual values of the type (`true` and `false`), the type `Boolean?` or `Nullable<Boolean>` additionally admits the `null` value. Tracking this in the type system is very useful: the type checker and other tooling can help programmers remember to check for null, and APIs that explicitly describe nullability through type signatures are more informative than ones that don't. However, these nullable types differ from Lean's `Option` in one very important way, which is that they don't allow multiple layers of optionality. `Option (Option Int)` can be constructed with `none`, `some none`, or `some (some 360)`. Kotlin, for example, treats `T??` as being equivalent to `T?`. This subtle difference is rarely relevant in practice, but it can matter from time to time.

To find the first entry in a list, if it exists, use `List.head?`. The question mark is part of the name, and is not related to the use of question marks to indicate nullable types in C# or Kotlin. In the definition of `List.head?`, an underscore is used to represent the tail of the list. In patterns, underscores match anything at all, but do not introduce variables to refer to the matched data. Using underscores instead of names is a way to clearly communicate to readers that part of the input is ignored.

```
def List.head? {α : Type} (xs : List α) : Option α :=
  match xs with
  | [] => none
  | y :: _ => some y
```

A Lean naming convention is to define operations that might fail in groups using the suffixes `?` for a version that returns an `Option`, `!` for a version that crashes when provided with invalid input, and `D` for a version that returns a default value when the operation would otherwise fail. Following this pattern, `head` requires the caller to provide mathematical evidence that the list is not empty, `head?` returns an `Option`, `head!` crashes the program when passed an empty list, and `headD` takes a default value to return in case the list is empty. The question mark and exclamation mark are part of the name, not special syntax, as Lean's naming rules are more liberal than many languages.

Because `head?` is defined in the `List` namespace, it can be used with accessor notation:

```
#eval primesUnder10.head?
```

```
some 2
```

However, attempting to test it on the empty list leads to two errors:

```
#eval [].head?
```

```
don't know how to synthesize implicit argument 'α'
  @List.nil ?m.18195
context:
⊢ Type ?u.18192

don't know how to synthesize implicit argument 'α'
  @_root_.List.head? ?m.18195 []
context:
⊢ Type ?u.18192
```

This is because Lean was unable to fully determine the expression's type. In particular, it could neither find the implicit type argument to `List.head?`, nor could it find the implicit type argument to `List.nil`. In Lean's output, `?m.XYZ` represents a part of a program that could not be inferred. These unknown parts are called *metavariables*, and they occur in some error messages. In order to evaluate an expression, Lean needs to be able to find its type, and the type was unavailable because the empty list does not have any entries from which the type can be found. Explicitly providing a type allows Lean to proceed:

```
#eval [].head? (α := Int)
```

```
none
```

The type can also be provided with a type annotation:

```
#eval ([] : List Int).head?
```

```
none
```

The error messages provide a useful clue. Both messages use the *same* metavariable to describe the missing implicit argument, which means that Lean has determined that the two missing pieces will share a solution, even though it was unable to determine the actual value of the solution.

`Prod`

The `Prod` structure, short for "Product", is a generic way of joining two values together. For instance, a `Prod Nat String` contains a `Nat` and a `String`. In other words, `PPoint Nat` could be replaced by `Prod Nat Nat`. `Prod` is very much like C#'s tuples, the `Pair` and `Triple` types in Kotlin, and `tuple` in C++. Many applications are best served by defining their own structures, even for simple cases like `Point`, because using domain terminology can make it easier to read the code. Additionally, defining structure types helps catch more errors by assigning different types to different domain concepts, preventing them from being mixed up.

On the other hand, there are some cases where it is not worth the overhead of defining a new type. Additionally, some libraries are sufficiently generic that there is no more specific concept than "pair". Finally, the standard library contains a variety of convenience functions that make it easier to work with the built-in pair type.

The standard pair structure is called `Prod`.

```
structure Prod (α : Type) (β : Type) : Type where
  fst : α
  snd : β
```

Lists are used so frequently that there is special syntax to make them more readable. For the same reason, both the product type and its constructor have special syntax. The type `Prod α β` is typically written `α × β`, mirroring the usual notation for a Cartesian product of sets. Similarly, the usual mathematical notation for pairs is available for `Prod`. In other words, instead of writing:

```
def fives : String × Int := { fst := "five", snd := 5 }
```

it suffices to write:

```
def fives : String × Int := ("five", 5)
```

Both notations are right-associative. This means that the following definitions are equivalent:

```
def sevens : String × Int × Nat := ("VII", 7, 4 + 3)
```

```
def sevens : String × (Int × Nat) := ("VII", (7, 4 + 3))
```

In other words, all products of more than two types, and their corresponding constructors, are actually nested products and nested pairs behind the scenes.

Sum

The `Sum` datatype is a generic way of allowing a choice between values of two different types. For instance, a `Sum String Int` is either a `String` or an `Int`. Like `Prod`, `Sum` should be used either when writing very generic code, for a very small section of code where there is no sensible domain-specific type, or when the standard library contains useful functions. In most situations, it is more readable and maintainable to use a custom inductive type.

Values of type `Sum α β` are either the constructor `inl` applied to a value of type `α` or the constructor `inr` applied to a value of type `β`:

```
inductive Sum (α : Type) (β : Type) : Type where
  | inl : α → Sum α β
  | inr : β → Sum α β
```

These names are abbreviations for "left injection" and "right injection", respectively. Just as the Cartesian product notation is used for `Prod`, a "circled plus" notation is used for `Sum`, so `α ⊕ β` is another way to write `Sum α β`. There is no special syntax for `Sum.inl` and `Sum.inr`.

As an example, if pet names can either be dog names or cat names, then a type for them can be introduced as a sum of strings:

```
def PetName : Type := String ⊕ String
```

In a real program, it would usually be better to define a custom inductive datatype for this purpose with informative constructor names. Here, `Sum.inl` is to be used for dog names, and `Sum.inr` is to be used for cat names. These constructors can be used to write a list of animal names:

```
def animals : List PetName :=
  [Sum.inl "Spot", Sum.inr "Tiger", Sum.inl "Fifi", Sum.inl "Rex", Sum.inr
"Floof"]
```

Pattern matching can be used to distinguish between the two constructors. For instance, a function that counts the number of dogs in a list of animal names (that is, the number of `Sum.inl` constructors) looks like this:

```
def howManyDogs (pets : List PetName) : Nat :=
  match pets with
  | [] => 0
  | Sum.inl _ :: morePets => howManyDogs morePets + 1
  | Sum.inr _ :: morePets => howManyDogs morePets
```

Function calls are evaluated before infix operators, so `howManyDogs morePets + 1` is the same as `(howManyDogs morePets) + 1`. As expected, `#eval howManyDogs animals` yields `3`.

Unit

`Unit` is a type with just one argumentless constructor, called `unit`. In other words, it describes only a single value, which consists of said constructor applied to no arguments whatsoever. `Unit` is defined as follows:

```
inductive Unit : Type where
  | unit : Unit
```

On its own, `Unit` is not particularly useful. However, in polymorphic code, it can be used as a placeholder for data that is missing. For instance, the following inductive datatype represents arithmetic expressions:

```
inductive ArithExpr (ann : Type) : Type where
  | int : ann → Int → ArithExpr ann
  | plus : ann → ArithExpr ann → ArithExpr ann → ArithExpr ann
  | minus : ann → ArithExpr ann → ArithExpr ann → ArithExpr ann
  | times : ann → ArithExpr ann → ArithExpr ann → ArithExpr ann
```

The type argument `ann` stands for annotations, and each constructor is annotated. Expressions coming from a parser might be annotated with source locations, so a return type of `ArithExpr SourcePos` ensures that the parser put a `SourcePos` at each subexpression. Expressions that don't come from the parser, however, will not have source locations, so their type can be `ArithExpr Unit`.

Additionally, because all Lean functions have arguments, zero-argument functions in other languages can be represented as functions that take a `Unit` argument. In a return position, the `Unit` type is similar to `void` in languages derived from C. In the C family, a function that returns `void` will return control to its caller, but it will not return any interesting value. By being an intentionally uninteresting value, `Unit` allows this to be expressed without requiring a special-purpose `void` feature in the type system. Unit's constructor can be written as empty parentheses: `() : Unit`.

Empty

The `Empty` datatype has no constructors whatsoever. Thus, it indicates unreachable code, because no series of calls can ever terminate with a value at type `Empty`.

`Empty` is not used nearly as often as `Unit`. However, it is useful in some specialized contexts. Many polymorphic datatypes do not use all of their type arguments in all of their constructors. For instance, `Sum.inl` and `Sum.inr` each use only one of `Sum`'s type arguments. Using `Empty` as one of the type arguments to `Sum` can rule out one of the constructors at a particular point in a program. This can allow generic code to be used in contexts that have additional restrictions.

Naming: Sums, Products, and Units

Generally speaking, types that offer multiple constructors are called *sum types*, while types whose single constructor takes multiple arguments are called *product types*. These terms are related to sums and products used in ordinary arithmetic. The relationship is easiest to see when the types involved contain a finite number of values. If `α` and `β` are types that contain n and k distinct values, respectively, then `α ⊕ β` contains $n + k$ distinct values and `α × β` contains $n \times k$ distinct values. For instance, `Bool` has two values: `true` and `false`, and `Unit` has one value: `Unit.unit`. The product `Bool × Unit` has the two values `(true, Unit.unit)` and `(false, Unit.unit)`, and the sum `Bool ⊕ Unit` has the three values `Sum.inl true`, `Sum.inl false`, and `Sum.inr unit`. Similarly, $2 \times 1 = 2$, and $2 + 1 = 3$.

Messages You May Meet

Not all definable structures or inductive types can have the type `Type`. In particular, if a constructor takes an arbitrary type as an argument, then the inductive type must have a different type. These errors usually state something about "universe levels". For example, for this inductive type:

```
inductive MyType : Type where
  | ctor : (α : Type) → α → MyType
```

Lean gives the following error:

```
invalid universe level in constructor 'MyType.ctor', parameter 'α' has type
  Type
at universe level
  2
which is not less than or equal to the inductive type's resulting universe level
  1
```

A later chapter describes why this is the case, and how to modify definitions to make them work. For now, try making the type an argument to the inductive type as a whole, rather than to the constructor.

Similarly, if a constructor's argument is a function that takes the datatype being defined as an argument, then the definition is rejected. For example:

```
inductive MyType : Type where
  | ctor : (MyType → Int) → MyType
```

yields the message:

```
(kernel) arg #1 of 'MyType.ctor' has a non positive occurrence of the datatypes
being declared
```

For technical reasons, allowing these datatypes could make it possible to undermine Lean's internal logic, making it unsuitable for use as a theorem prover.

Recursive functions that take two parameters should not match against the pair, but rather match each parameter independently. Otherwise, the mechanism in Lean that checks whether recursive calls are made on smaller values is unable to see the connection between the input value and the argument in the recursive call. For example, this function that determines whether two lists have the same length is rejected:

```
def sameLength (xs : List α) (ys : List β) : Bool :=
  match (xs, ys) with
  | ([], []) => true
  | (x :: xs', y :: ys') => sameLength xs' ys'
  | _ => false
```

The error message is:

```
fail to show termination for
  sameLength
with errors
failed to infer structural recursion:
Not considering parameter α of sameLength:
  it is unchanged in the recursive calls
Not considering parameter β of sameLength:
  it is unchanged in the recursive calls
Cannot use parameter xs:
  failed to eliminate recursive application
    sameLength xs' ys'
Cannot use parameter ys:
  failed to eliminate recursive application
    sameLength xs' ys'


Could not find a decreasing measure.
The basic measures relate at each recursive call as follows:
(<, ≤, =: relation proved, ? all proofs failed, _: no proof attempted)
              xs ys
1) 1282:30-48  ?  ?
Please use `termination_by` to specify a decreasing measure.
```

The problem can be fixed through nested pattern matching:

```
def sameLength (xs : List α) (ys : List β) : Bool :=
  match xs with
  | [] =>
    match ys with
    | [] => true
    | _ => false
  | x :: xs' =>
    match ys with
    | y :: ys' => sameLength xs' ys'
    | _ => false
```

Simultaneous matching, described in the next section, is another way to solve the problem that is often more elegant.

Forgetting an argument to an inductive type can also yield a confusing message. For example, when the argument `α` is not passed to `MyType` in `ctor`'s type:

```
inductive MyType (α : Type) : Type where
  | ctor : α → MyType
```

Lean replies with the following error:

```
type expected, got
  (MyType : Type → Type)
```

The error message is saying that `MyType`'s type, which is `Type → Type`, does not itself describe types. `MyType` requires an argument to become an actual honest-to-goodness type.

The same message can appear when type arguments are omitted in other contexts, such as in a type signature for a definition:

```
inductive MyType (α : Type) : Type where
  | ctor : α → MyType α

def ofFive : MyType := ctor 5
```

Exercises

- Write a function to find the last entry in a list. It should return an `Option`.
- Write a function that finds the first entry in a list that satisfies a given predicate. Start the definition with `def List.findFirst? {α : Type} (xs : List α) (predicate : α → Bool) : Option α :=`

- Write a function `Prod.switch` that switches the two fields in a pair for each other. Start the definition with `def Prod.switch {α β : Type} (pair : α × β) : β × α :=`
- Rewrite the `PetName` example to use a custom datatype and compare it to the version that uses `Sum`.
- Write a function `zip` that combines two lists into a list of pairs. The resulting list should be as long as the shortest input list. Start the definition with `def zip {α β : Type} (xs : List α) (ys : List β) : List (α × β) :=`.
- Write a polymorphic function `take` that returns the first n entries in a list, where n is a `Nat`. If the list contains fewer than `n` entries, then the resulting list should be the entire input list. `#eval take 3 ["bolete", "oyster"]` should yield `["bolete", "oyster"]`, and `#eval take 1 ["bolete", "oyster"]` should yield `["bolete"]`.
- Using the analogy between types and arithmetic, write a function that distributes products over sums. In other words, it should have type `α × (β ⊕ γ) → (α × β) ⊕ (α × γ)`.
- Using the analogy between types and arithmetic, write a function that turns multiplication by two into a sum. In other words, it should have type `Bool × α → α ⊕ α`.

Additional Conveniences

Lean contains a number of convenience features that make programs much more concise.

Automatic Implicit Parameters

When writing polymorphic functions in Lean, it is typically not necessary to list all the implicit parameters. Instead, they can simply be mentioned. If Lean can determine their type, then they are automatically inserted as implicit parameters. In other words, the previous definition of `length`:

```
def length {α : Type} (xs : List α) : Nat :=
  match xs with
  | [] => 0
  | y :: ys => Nat.succ (length ys)
```

can be written without `{α : Type}`:

```
def length (xs : List α) : Nat :=
  match xs with
  | [] => 0
  | y :: ys => Nat.succ (length ys)
```

This can greatly simplify highly polymorphic definitions that take many implicit parameters.

Pattern-Matching Definitions

When defining functions with `def`, it is quite common to name an argument and then immediately use it with pattern matching. For instance, in `length`, the argument `xs` is used only in `match`. In these situations, the cases of the `match` expression can be written directly, without naming the argument at all.

The first step is to move the arguments' types to the right of the colon, so the return type is a function type. For instance, the type of `length` is `List α → Nat`. Then, replace the `:=` with each case of the pattern match:

```
def length : List α → Nat
  | [] => 0
  | y :: ys => Nat.succ (length ys)
```

This syntax can also be used to define functions that take more than one argument. In this case, their patterns are separated by commas. For instance, `drop` takes a number n and a list, and returns the list after removing the first n entries.

```
def drop : Nat → List α → List α
  | Nat.zero, xs => xs
  | _, [] => []
  | Nat.succ n, x :: xs => drop n xs
```

Named arguments and patterns can also be used in the same definition. For instance, a function that takes a default value and an optional value, and returns the default when the optional value is `none`, can be written:

```
def fromOption (default : α) : Option α → α
  | none => default
  | some x => x
```

This function is called `Option.getD` in the standard library, and can be called with dot notation:

```
#eval (some "salmonberry").getD ""
```

```
"salmonberry"
```

```
#eval none.getD ""
```

```
""
```

Local Definitions

It is often useful to name intermediate steps in a computation. In many cases, intermediate values represent useful concepts all on their own, and naming them explicitly can make the program easier to read. In other cases, the intermediate value is used more than once. As in most other languages, writing down the same code twice in Lean causes it to be computed twice, while saving the result in a variable leads to the result of the computation being saved and re-used.

For instance, `unzip` is a function that transforms a list of pairs into a pair of lists. When the list of pairs is empty, then the result of `unzip` is a pair of empty lists. When the list of pairs has a pair at its head, then the two fields of the pair are added to the result of unzipping the rest of the list. This definition of `unzip` follows that description exactly:

```
def unzip : List (α × β) → List α × List β
  | [] => ([], [])
  | (x, y) :: xys =>
    (x :: (unzip xys).fst, y :: (unzip xys).snd)
```

Unfortunately, there is a problem: this code is slower than it needs to be. Each entry in the list of pairs leads to two recursive calls, which makes this function take exponential time. However, both recursive calls will have the same result, so there is no reason to make the recursive call twice.

In Lean, the result of the recursive call can be named, and thus saved, using `let`. Local definitions with `let` resemble top-level definitions with `def`: it takes a name to be locally defined, arguments if desired, a type signature, and then a body following `:=`. After the local definition, the expression in which the local definition is available (called the *body* of the `let`-expression) must be on a new line, starting at a column in the file that is less than or equal to that of the `let` keyword. A local definition with `let` in `unzip` looks like this:

```
def unzip : List (α × β) → List α × List β
  | [] => ([], [])
  | (x, y) :: xys =>
    let unzipped : List α × List β := unzip xys
    (x :: unzipped.fst, y :: unzipped.snd)
```

To use `let` on a single line, separate the local definition from the body with a semicolon.

Local definitions with `let` may also use pattern matching when one pattern is enough to match all cases of a datatype. In the case of `unzip`, the result of the recursive call is a pair. Because pairs have only a single constructor, the name `unzipped` can be replaced with a pair pattern:

```
def unzip : List (α × β) → List α × List β
  | [] => ([], [])
  | (x, y) :: xys =>
    let (xs, ys) : List α × List β := unzip xys
    (x :: xs, y :: ys)
```

Judicious use of patterns with `let` can make code easier to read, compared to writing the accessor calls by hand.

The biggest difference between `let` and `def` is that recursive `let` definitions must be explicitly indicated by writing `let rec`. For instance, one way to reverse a list involves a recursive helper function, as in this definition:

```
def reverse (xs : List α) : List α :=
  let rec helper : List α → List α → List α
    | [], soFar => soFar
    | y :: ys, soFar => helper ys (y :: soFar)
  helper xs []
```

The helper function walks down the input list, moving one entry at a time over to `soFar` . When it reaches the end of the input list, `soFar` contains a reversed version of the input.

Type Inference

In many situations, Lean can automatically determine an expression's type. In these cases, explicit types may be omitted from both top-level definitions (with `def`) and local definitions (with `let`). For example, the recursive call to `unzip` does not need an annotation:

```
def unzip : List (α × β) → List α × List β
  | [] => ([], [])
  | (x, y) :: xys =>
    let unzipped := unzip xys
    (x :: unzipped.fst, y :: unzipped.snd)
```

As a rule of thumb, omitting the types of literal values (like strings and numbers) usually works, although Lean may pick a type for literal numbers that is more specific than the intended type. Lean can usually determine a type for a function application, because it already knows the argument types and the return type. Omitting return types for function definitions will often work, but function parameters typically require annotations. Definitions that are not functions, like `unzipped` in the example, do not need type annotations if their bodies do not need type annotations, and the body of this definition is a function application.

Omitting the return type for `unzip` is possible when using an explicit `match` expression:

```
def unzip (pairs : List (α × β)) :=
  match pairs with
  | [] => ([], [])
  | (x, y) :: xys =>
    let unzipped := unzip xys
    (x :: unzipped.fst, y :: unzipped.snd)
```

Generally speaking, it is a good idea to err on the side of too many, rather than too few, type annotations. First off, explicit types communicate assumptions about the code to readers. Even if Lean can determine the type on its own, it can still be easier to read code without having to repeatedly query Lean for type information. Secondly, explicit types help localize errors. The more explicit a program is about its types, the more informative the error messages can be. This is especially important in a language like Lean that has a very expressive type system. Thirdly, explicit types make it easier to write the program in the first place. The type is a specification, and the compiler's feedback can be a helpful tool in writing a program that meets the specification. Finally, Lean's type inference is a best-effort system. Because Lean's type system is so expressive, there is no "best" or most general type to find for all expressions. This means that even if you get a type, there's no guarantee that it's the *right* type for a given application. For instance, `14` can be a `Nat` or an `Int`:

```
#check 14
```

```
14 : Nat
```

```
#check (14 : Int)
```

```
14 : Int
```

Missing type annotations can give confusing error messages. Omitting all types from the definition of `unzip`:

```
def unzip pairs :=
  match pairs with
  | [] => ([], [])
  | (x, y) :: xys =>
    let unzipped := unzip xys
    (x :: unzipped.fst, y :: unzipped.snd)
```

leads to a message about the `match` expression:

```
invalid match-expression, pattern contains metavariables
  []
```

This is because `match` needs to know the type of the value being inspected, but that type was not available. A "metavariable" is an unknown part of a program, written `?m.XYZ` in error messages—they are described in the section on Polymorphism. In this program, the type annotation on the argument is required.

Even some very simple programs require type annotations. For instance, the identity function just returns whatever argument it is passed. With argument and type annotations, it looks like this:

```
def id (x : α) : α := x
```

Lean is capable of determining the return type on its own:

```
def id (x : α) := x
```

Omitting the argument type, however, causes an error:

```
def id x := x
```

```
failed to infer binder type
```

In general, messages that say something like "failed to infer" or that mention metavariables are often a sign that more type annotations are necessary. Especially while still learning Lean, it is useful to provide most types explicitly.

Simultaneous Matching

Pattern-matching expressions, just like pattern-matching definitions, can match on multiple values at once. Both the expressions to be inspected and the patterns that they match against are written with commas between them, similarly to the syntax used for definitions. Here is a version of `drop` that uses simultaneous matching:

```
def drop (n : Nat) (xs : List α) : List α :=
  match n, xs with
  | Nat.zero, ys => ys
  | _, [] => []
  | Nat.succ n , y :: ys => drop n ys
```

Simultaneous matching resembles matching on a pair, but there is an important difference. Lean tracks the connection between the expression being matched and the patterns, and this information is used for purposes that include checking for termination and propagating static type information. As a result, the version of `sameLength` that matches a pair is rejected by the termination checker, because the connection between `xs` and `x :: xs'` is obscured by the intervening pair:

```
def sameLength (xs : List α) (ys : List β) : Bool :=
  match (xs, ys) with
  | ([], []) => true
  | (x :: xs', y :: ys') => sameLength xs' ys'
  | _ => false
```

```
fail to show termination for
  sameLength
with errors
failed to infer structural recursion:
Not considering parameter α of sameLength:
  it is unchanged in the recursive calls
Not considering parameter β of sameLength:
  it is unchanged in the recursive calls
Cannot use parameter xs:
  failed to eliminate recursive application
    sameLength xs' ys'
Cannot use parameter ys:
  failed to eliminate recursive application
    sameLength xs' ys'


Could not find a decreasing measure.
The basic measures relate at each recursive call as follows:
(<, ≤, =: relation proved, ? all proofs failed, _: no proof attempted)
              xs ys
1) 1282:30-48  ?  ?
Please use `termination_by` to specify a decreasing measure.
```

Simultaneously matching both lists is accepted:

```
def sameLength (xs : List α) (ys : List β) : Bool :=
  match xs, ys with
  | [], [] => true
  | x :: xs', y :: ys' => sameLength xs' ys'
  | _, _ => false
```

Natural Number Patterns

In the section on datatypes and patterns, `even` was defined like this:

```
def even (n : Nat) : Bool :=
  match n with
  | Nat.zero => true
  | Nat.succ k => not (even k)
```

Just as there is special syntax to make list patterns more readable than using `List.cons` and `List.nil` directly, natural numbers can be matched using literal numbers and `+`. For example, `even` can also be defined like this:

```
def even : Nat → Bool
  | 0 => true
  | n + 1 => not (even n)
```

In this notation, the arguments to the `+` pattern serve different roles. Behind the scenes, the left argument (`n` above) becomes an argument to some number of `Nat.succ` patterns, and the right argument (`1` above) determines how many `Nat.succ`s to wrap around the pattern. The explicit patterns in `halve`, which divides a `Nat` by two and drops the remainder:

```
def halve : Nat → Nat
  | Nat.zero => 0
  | Nat.succ Nat.zero => 0
  | Nat.succ (Nat.succ n) => halve n + 1
```

can be replaced by numeric literals and `+`:

```
def halve : Nat → Nat
  | 0 => 0
  | 1 => 0
  | n + 2 => halve n + 1
```

Behind the scenes, both definitions are completely equivalent. Remember: `halve n + 1` is equivalent to `(halve n) + 1`, not `halve (n + 1)`.

When using this syntax, the second argument to `+` should always be a literal `Nat`. Even though addition is commutative, flipping the arguments in a pattern can result in errors like the following:

```
def halve : Nat → Nat
  | 0 => 0
  | 1 => 0
  | 2 + n => halve n + 1
```

```
invalid patterns, `n` is an explicit pattern variable, but it only occurs in
positions that are inaccessible to pattern matching
  .(Nat.add 2 n)
```

This restriction enables Lean to transform all uses of the `+` notation in a pattern into uses of the underlying `Nat.succ`, keeping the language simpler behind the scenes.

Anonymous Functions

Functions in Lean need not be defined at the top level. As expressions, functions are produced with the `fun` syntax. Function expressions begin with the keyword `fun`, followed by one or more parameters, which are separated from the return expression using `=>`. For instance, a function that adds one to a number can be written:

```
#check fun x => x + 1
```

```
fun x => x + 1 : Nat → Nat
```

Type annotations are written the same way as on `def`, using parentheses and colons:

```
#check fun (x : Int) => x + 1
```

```
fun x => x + 1 : Int → Int
```

Similarly, implicit parameters may be written with curly braces:

```
#check fun {α : Type} (x : α) => x
```

```
fun {α} x => x : {α : Type} → α → α
```

This style of anonymous function expression is often referred to as a *lambda expression*, because the typical notation used in mathematical descriptions of programming languages uses the Greek letter λ (lambda) where Lean has the keyword `fun`. Even though Lean does permit `λ` to be used instead of `fun`, it is most common to write `fun`.

Anonymous functions also support the multiple-pattern style used in `def`. For instance, a function that returns the predecessor of a natural number if it exists can be written:

```
#check fun
  | 0 => none
  | n + 1 => some n
```

```
fun x =>
  match x with
  | 0 => none
  | n.succ => some n : Nat → Option Nat
```

Note that Lean's own description of the function has a named argument and a `match` expression. Many of Lean's convenient syntactic shorthands are expanded to simpler syntax behind the scenes, and the abstraction sometimes leaks.

Definitions using `def` that take arguments may be rewritten as function expressions. For instance, a function that doubles its argument can be written as follows:

```
def double : Nat → Nat := fun
  | 0 => 0
  | k + 1 => double k + 2
```

When an anonymous function is very simple, like `fun x => x + 1`, the syntax for creating the function can be fairly verbose. In that particular example, six non-whitespace characters are used to introduce the function, and its body consists of only three non-whitespace characters. For these simple cases, Lean provides a shorthand. In an expression surrounded by parentheses, a centered dot character `·` can stand for an parameter, and the expression inside the parentheses becomes the function's body. That particular function can also be written `(· + 1)`.

The centered dot always creates a function out of the *closest* surrounding set of parentheses. For instance, `(· + 5, 3)` is a function that returns a pair of numbers, while `((· + 5), 3)` is a pair of a function and a number. If multiple dots are used, then they become parameters from left to right:

```
(· , ·) 1 2
===>
(1, ·) 2
===>
(1, 2)
```

Anonymous functions can be applied in precisely the same way as functions defined using `def` or `let`. The command `#eval (fun x => x + x) 5` results in:

```
10
```

while `#eval (· * 2) 5` results in:

```
10
```

Namespaces

Each name in Lean occurs in a *namespace*, which is a collection of names. Names are placed in namespaces using `.`, so `List.map` is the name `map` in the `List` namespace. Names in different namespaces do not conflict with each other, even if they are otherwise identical. This means that `List.map` and `Array.map` are different names. Namespaces may be nested, so `Project.Frontend.User.loginTime` is the name `loginTime` in the nested namespace `Project.Frontend.User` .

Names can be directly defined within a namespace. For instance, the name `double` can be defined in the `Nat` namespace:

```
def Nat.double (x : Nat) : Nat := x + x
```

Because `Nat` is also the name of a type, dot notation is available to call `Nat.double` on expressions with type `Nat` :

```
#eval (4 : Nat).double
```

```
8
```

In addition to defining names directly in a namespace, a sequence of declarations can be placed in a namespace using the `namespace` and `end` commands. For instance, this defines `triple` and `quadruple` in the namespace `NewNamespace` :

```
namespace NewNamespace
def triple (x : Nat) : Nat := 3 * x
def quadruple (x : Nat) : Nat := 2 * x + 2 * x
end NewNamespace
```

To refer to them, prefix their names with `NewNamespace.` :

```
#check NewNamespace.triple
```

```
NewNamespace.triple (x : Nat) : Nat
```

```
#check NewNamespace.quadruple
```

```
NewNamespace.quadruple (x : Nat) : Nat
```

Namespaces may be *opened*, which allows the names in them to be used without explicit qualification. Writing `open MyNamespace in` before an expression causes the contents of `MyNamespace` to be available in the expression. For example, `timesTwelve` uses both `quadruple` and `triple` after opening `NewNamespace` :

```
def timesTwelve (x : Nat) :=
  open NewNamespace in
  quadruple (triple x)
```

Namespaces can also be opened prior to a command. This allows all parts of the command to refer to the contents of the namespace, rather than just a single expression. To do this, place the `open ... in` prior to the command.

```
open NewNamespace in
#check quadruple
```

```
NewNamespace.quadruple (x : Nat) : Nat
```

Function signatures show the name's full namespace. Namespaces may additionally be opened for *all* following commands for the rest of the file. To do this, simply omit the `in` from a top-level usage of `open`.

if let

When consuming values that have a sum type, it is often the case that only a single constructor is of interest. For example, given this type that represents a subset of Markdown inline elements:

```
inductive Inline : Type where
  | lineBreak
  | string : String → Inline
  | emph : Inline → Inline
  | strong : Inline → Inline
```

a function that recognizes string elements and extracts their contents can be written:

```
def Inline.string? (inline : Inline) : Option String :=
  match inline with
  | Inline.string s => some s
  | _ => none
```

An alternative way of writing this function's body uses `if` together with `let`:

```
def Inline.string? (inline : Inline) : Option String :=
  if let Inline.string s := inline then
    some s
  else none
```

This is very much like the pattern-matching `let` syntax. The difference is that it can be used with sum types, because a fallback is provided in the `else` case. In some contexts, using `if let` instead of `match` can make code easier to read.

Positional Structure Arguments

The section on structures presents two ways of constructing structures:

1. The constructor can be called directly, as in `Point.mk 1 2`.
2. Brace notation can be used, as in `{ x := 1, y := 2 }`.

In some contexts, it can be convenient to pass arguments positionally, rather than by name, but without naming the constructor directly. For instance, defining a variety of similar structure types can help keep domain concepts separate, but the natural way to read the code may treat each of them as being essentially a tuple. In these contexts, the arguments can be enclosed in angle brackets `⟨` and `⟩`. A `Point` can be written `⟨1, 2⟩`. Be careful! Even though they look like the less-than sign `<` and greater-than sign `>`, these brackets are different. They can be input using `\<` and `\>`, respectively.

Just as with the brace notation for named constructor arguments, this positional syntax can only be used in a context where Lean can determine the structure's type, either from a type annotation or from other type information in the program. For instance, `#eval ⟨1, 2⟩` yields the following error:

```
invalid constructor ⟨...⟩, expected type must be an inductive type 
  ?m.29916
```

The metavariable in the error is because there is no type information available. Adding an annotation, such as in `#eval (⟨1, 2⟩ : Point)`, solves the problem:

```
{ x := 1.000000, y := 2.000000 }
```

String Interpolation

In Lean, prefixing a string with `s!` triggers *interpolation*, where expressions contained in curly braces inside the string are replaced with their values. This is similar to `f`-strings in Python and `$`-prefixed strings in C#. For instance,

```
#eval s!"three fives is {NewNamespace.triple 5}"
```

yields the output

```
"three fives is 15"
```

Not all expressions can be interpolated into a string. For instance, attempting to interpolate a function results in an error.

```
#check s!"three fives is {NewNamespace.triple}"
```

yields the error

```
failed to synthesize
  ToString (Nat → Nat)

Additional diagnostic information may be available using the `set_option
diagnostics true` command.
```

This is because there is no standard way to convert functions into strings. The Lean compiler maintains a table that describes how to convert values of various types into strings, and the message `failed to synthesize instance` means that the Lean compiler didn't find an entry in this table for the given type. This uses the same language feature as the `deriving Repr` syntax that was described in the section on structures.

Summary

Evaluating Expressions

In Lean, computation occurs when expressions are evaluated. This follows the usual rules of mathematical expressions: sub-expressions are replaced by their values following the usual order of operations, until the entire expression has become a value. When evaluating an `if` or a `match`, the expressions in the branches are not evaluated until the value of the condition or the match subject has been found.

Once they have been given a value, variables never change. Similarly to mathematics but unlike most programming languages, Lean variables are simply placeholders for values, rather than addresses to which new values can be written. Variables' values may come from global definitions with `def`, local definitions with `let`, as named arguments to functions, or from pattern matching.

Functions

Functions in Lean are first-class values, meaning that they can be passed as arguments to other functions, saved in variables, and used like any other value. Every Lean function takes exactly one argument. To encode a function that takes more than one argument, Lean uses a technique called currying, where providing the first argument returns a function that expects the remaining arguments. To encode a function that takes no arguments, Lean uses the `Unit` type, which is the least informative possible argument.

There are three primary ways of creating functions:

1. Anonymous functions are written using `fun`. For instance, a function that swaps the fields of a `Point` can be written `fun (point : Point) => { x := point.y, y := point.x : Point }`
2. Very simple anonymous functions are written by placing one or more centered dots `·` inside of parentheses. Each centered dot becomes an argument to the function, and the parentheses delimit its body. For instance, a function that subtracts one from its argument can be written as `(· - 1)` instead of as `fun x => x - 1`.
3. Functions can be defined using `def` or `let` by adding an argument list or by using pattern-matching notation.

Types

Lean checks that every expression has a type. Types, such as `Int`, `Point`, `{α : Type} → Nat → α → List α`, and `Option (String ⊕ (Nat × String))`, describe the values that may eventually be found for an expression. Like other languages, types in Lean can express lightweight specifications for programs that are checked by the Lean compiler, obviating the need for certain classes of unit test. Unlike most languages, Lean's types can also express arbitrary mathematics, unifying the worlds of programming and theorem proving. While using Lean for proving theorems is mostly out of scope for this book, *Theorem Proving in Lean 4* contains more information on this topic.

Some expressions can be given multiple types. For instance, `3` can be an `Int` or a `Nat`. In Lean, this should be understood as two separate expressions, one with type `Nat` and one with type `Int`, that happen to be written in the same way, rather than as two different types for the same thing.

Lean is sometimes able to determine types automatically, but types must often be provided by the user. This is because Lean's type system is so expressive. Even when Lean can find a type, it may not find the desired type— `3` could be intended to be used as an `Int`, but Lean will give it the type `Nat` if there are no further constraints. In general, it is a good idea to write most types explicitly, only letting Lean fill out the very obvious types. This improves Lean's error messages and helps make programmer intent more clear.

Some functions or datatypes take types as arguments. They are called *polymorphic*. Polymorphism allows programs such as one that calculates the length of a list without caring what type the entries in the list have. Because types are first class in Lean, polymorphism does not require any special syntax, so types are passed just like other arguments. Naming an argument in a function type allows later types to mention that name, and when the function is applied to an argument, the type of the resulting term is found by replacing the argument's name with the actual value it was applied to.

Structures and Inductive Types

Brand new datatypes can be introduced to Lean using the `structure` or `inductive` features. These new types are not considered to be equivalent to any other type, even if their definitions are otherwise identical. Datatypes have *constructors* that explain the ways in which their values can be constructed, and each constructor takes some number of arguments. Constructors in Lean are not the same as constructors in object-oriented languages: Lean's constructors are inert holders of data, rather than active code that initializes an allocated object.

Typically, `structure` is used to introduce a product type (that is, a type with just one constructor that takes any number of arguments), while `inductive` is used to introduce a sum type (that is, a type with many distinct constructors). Datatypes defined with `structure` are provided with one accessor function for each field. Both structures and inductive datatypes may be consumed with pattern matching, which exposes the values stored inside of constructors using a subset of the syntax used to call said constructors. Pattern matching means that knowing how to create a value implies knowing how to consume it.

Recursion

A definition is recursive when the name being defined is used in the definition itself. Because Lean is an interactive theorem prover in addition to being a programming language, there are certain restrictions placed on recursive definitions. In Lean's logical side, circular definitions could lead to logical inconsistency.

In order to ensure that recursive definitions do not undermine the logical side of Lean, Lean must be able to prove that all recursive functions terminate, no matter what arguments they are called with. In practice, this means either that recursive calls are all performed on a structurally-smaller piece of the input, which ensures that there is always progress towards a base case, or that users must provide some other evidence that the function always terminates. Similarly, recursive inductive types are not allowed to have a constructor that takes a function *from* the type as an argument, because this would make it possible to encode non-terminating functions.

Hello, World!

While Lean has been designed to have a rich interactive environment in which programmers can get quite a lot of feedback from the language without leaving the confines of their favorite text editor, it is also a language in which real programs can be written. This means that it also has a batch-mode compiler, a build system, a package manager, and all the other tools that are necessary for writing programs.

While the previous chapter presented the basics of functional programming in Lean, this chapter explains how to start a programming project, compile it, and run the result. Programs that run and interact with their environment (e.g. by reading input from standard input or creating files) are difficult to reconcile with the understanding of computation as the evaluation of mathematical expressions. In addition to a description of the Lean build tools, this chapter also provides a way to think about functional programs that interact with the world.

Running a Program

The simplest way to run a Lean program is to use the `--run` option to the Lean executable. Create a file called `Hello.lean` and enter the following contents:

```
def main : IO Unit := IO.println "Hello, world!"
```

Then, from the command line, run:

```
lean --run Hello.lean
```

The program displays `Hello, world!` and exits.

Anatomy of a Greeting

When Lean is invoked with the `--run` option, it invokes the program's `main` definition. In programs that do not take command-line arguments, `main` should have type `IO Unit`. This means that `main` is not a function, because there are no arrows (→) in its type. Instead of being a function that has side effects, `main` consists of a description of effects to be carried out.

As discussed in the preceding chapter, `Unit` is the simplest inductive type. It has a single constructor called `unit` that takes no arguments. Languages in the C tradition have a notion of a `void` function that does not return any value at all. In Lean, all functions take an argument and return a value, and the lack of interesting arguments or return values can be signaled by using the `Unit` type instead. If `Bool` represents a single bit of information, `Unit` represents zero bits of information.

`IO α` is the type of a program that, when executed, will either throw an exception or return a value of type `α`. During execution, this program may have side effects. These programs are referred to as `IO` *actions*. Lean distinguishes between *evaluation* of expressions, which strictly adheres to the mathematical model of substitution of values for variables and reduction of sub-expressions without side effects, and *execution* of `IO` actions, which rely on an external system to interact with the world. `IO.println` is a function from strings to `IO` actions that, when executed, write the given string to standard output. Because this action doesn't read any interesting information from the environment in the process of emitting the string, `IO.println` has type `String → IO Unit`. If it did return something interesting, then that would be indicated by the `IO` action having a type other than `Unit`.

Functional Programming vs Effects

Lean's model of computation is based on the evaluation of mathematical expressions, in which variables are given exactly one value that does not change over time. The result of evaluating an expression does not change, and evaluating the same expression again will always yield the same result.

On the other hand, useful programs must interact with the world. A program that performs neither input nor output can't ask a user for data, create files on disk, or open network connections. Lean is written in itself, and the Lean compiler certainly reads files, creates files, and interacts with text editors. How can a language in which the same expression always yields the same result support programs that read files from disk, when the contents of these files might change over time?

This apparent contradiction can be resolved by thinking a bit differently about side effects. Imagine a café that sells coffee and sandwiches. This café has two employees: a cook who fulfills orders, and a worker at the counter who interacts with customers and places order slips. The cook is a surly person, who really prefers not to have any contact with the world outside, but who is very good at consistently delivering the food and drinks that the café is known for. In order to do this, however, the cook needs peace and quiet, and can't be disturbed with conversation. The counter worker is friendly, but completely incompetent in the kitchen. Customers interact with the counter worker, who delegates all actual cooking to the cook. If the cook has a question for a customer, such as clarifying an allergy, they send a little note to the counter worker, who interacts with the customer and passes a note back to the cook with the result.

In this analogy, the cook is the Lean language. When provided with an order, the cook faithfully and consistently delivers what is requested. The counter worker is the surrounding run-time system that interacts with the world and can accept payments, dispense food, and have conversations with customers. Working together, the two employees serve all the functions of the restaurant, but their responsibilities are divided, with each performing the tasks that they're best at. Just as keeping customers away allows the cook to focus on making truly excellent coffee and sandwiches, Lean's lack of side effects allows programs to be used as part of formal mathematical proofs. It also helps programmers understand the parts of the program in isolation from each other, because there are no hidden state changes that create subtle coupling between components. The cook's notes represent `IO` actions that are produced by evaluating Lean expressions, and the counter worker's replies are the values that are passed back from effects.

This model of side effects is quite similar to how the overall aggregate of the Lean language, its compiler, and its run-time system (RTS) work. Primitives in the run-time system, written in C, implement all the basic effects. When running a program, the RTS invokes the `main` action, which returns new `IO` actions to the RTS for execution. The RTS executes these actions, delegating to the user's Lean code to carry out computations. From the internal perspective of Lean, programs are free of side effects, and `IO` actions are just descriptions of tasks to be carried out. From the external perspective of the program's user, there is a layer of side effects that create an interface to the program's core logic.

Real-World Functional Programming

The other useful way to think about side effects in Lean is by considering `IO` actions to be functions that take the entire world as an argument and return a value paired with a new world. In this case, reading a line of text from standard input *is* a pure function, because a different world is provided as an argument each time. Writing a line of text to standard output is a pure function, because the world that the function returns is different from the one that it began with. Programs do need to be careful to never re-use the world, nor to fail to return a new world—this would amount to time travel or the end of the world, after all. Careful abstraction boundaries can make this style of programming safe. If every primitive `IO` action accepts one world and returns a new one, and they can only be combined with tools that preserve this invariant, then the problem cannot occur.

This model cannot be implemented. After all, the entire universe cannot be turned into a Lean value and placed into memory. However, it is possible to implement a variation of this model with an abstract token that stands for the world. When the program is started, it is provided with a world token. This token is then passed on to the IO primitives, and their returned tokens are similarly passed to the next step. At the end of the program, the token is returned to the operating system.

This model of side effects is a good description of how `IO` actions as descriptions of tasks to be carried out by the RTS are represented internally in Lean. The actual functions that transform the real world are behind an abstraction barrier. But real programs typically consist of a sequence of effects, rather than just one. To enable programs to use multiple effects, there is a sub-language of Lean called `do` notation that allows these primitive `IO` actions to be safely composed into a larger, useful program.

Combining `IO` Actions

Most useful programs accept input in addition to producing output. Furthermore, they may take decisions based on input, using the input data as part of a computation. The following program, called `HelloName.lean`, asks the user for their name and then greets them:

```
def main : IO Unit := do
  let stdin ← IO.getStdin
  let stdout ← IO.getStdout

  stdout.putStrLn "How would you like to be addressed?"
  let input ← stdin.getLine
  let name := input.dropRightWhile Char.isWhitespace

  stdout.putStrLn s!"Hello, {name}!"
```

In this program, the `main` action consists of a `do` block. This block contains a sequence of *statements*, which can be both local variables (introduced using `let`) and actions that are to be executed. Just as SQL can be thought of as a special-purpose language for interacting with databases, the `do` syntax can be thought of as a special-purpose sub-language within Lean that is dedicated to modeling imperative programs. `IO` actions that are built with a `do` block are executed by executing the statements in order.

This program can be run in the same manner as the prior program:

```
lean --run HelloName.lean
```

If the user responds with `David`, a session of interaction with the program reads:

```
How would you like to be addressed?
David
Hello, David!
```

The type signature line is just like the one for `Hello.lean`:

```
def main : IO Unit := do
```

The only difference is that it ends with the keyword `do`, which initiates a sequence of commands. Each indented line following the keyword `do` is part of the same sequence of commands.

The first two lines, which read:

```
  let stdin ← IO.getStdin
  let stdout ← IO.getStdout
```

retrieve the `stdin` and `stdout` handles by executing the library actions `IO.getStdin` and `IO.getStdout`, respectively. In a `do` block, `let` has a slightly different meaning than in an ordinary expression. Ordinarily, the local definition in a `let` can be used in just one expression, which immediately follows the local definition. In a `do` block, local bindings introduced by `let` are available in all statements in the remainder of the `do` block, rather than just the next one. Additionally, `let` typically connects the name being defined to its definition using `:=`, while some `let` bindings in `do` use a left arrow (`←` or `<-`) instead. Using an arrow means that the value of the expression is an `IO` action that should be executed, with the result of the action saved in the local variable. In other words, if the expression to the right of the arrow has type `IO α`, then the variable has type `α` in the remainder of the `do` block. `IO.getStdin` and `IO.getStdout` are `IO` actions in order to allow `stdin` and `stdout` to be locally overridden in a program, which can be convenient. If they were global variables as in C, then there would be no meaningful way to override them, but `IO` actions can return different values each time they are executed.

The next part of the `do` block is responsible for asking the user for their name:

```
stdout.putStrLn "How would you like to be addressed?"
let input ← stdin.getLine
let name := input.dropRightWhile Char.isWhitespace
```

The first line writes the question to `stdout`, the second line requests input from `stdin`, and the third line removes the trailing newline (plus any other trailing whitespace) from the input line. The definition of `name` uses `:=`, rather than `←`, because `String.dropRightWhile` is an ordinary function on strings, rather than an `IO` action.

Finally, the last line in the program is:

```
stdout.putStrLn s!"Hello, {name}!"
```

It uses string interpolation to insert the provided name into a greeting string, writing the result to `stdout`.

Step By Step

A `do` block can be executed one line at a time. Start with the program from the prior section:

```
let stdin ← IO.getStdin
let stdout ← IO.getStdout
stdout.putStrLn "How would you like to be addressed?"
let input ← stdin.getLine
let name := input.dropRightWhile Char.isWhitespace
stdout.putStrLn s!"Hello, {name}!"
```

Standard IO

The first line is `let stdin ← IO.getStdin`, while the remainder is:

```
let stdout ← IO.getStdout
stdout.putStrLn "How would you like to be addressed?"
let input ← stdin.getLine
let name := input.dropRightWhile Char.isWhitespace
stdout.putStrLn s!"Hello, {name}!"
```

To execute a `let` statement that uses a `←`, start by evaluating the expression to the right of the arrow (in this case, `IO.getStdIn`). Because this expression is just a variable, its value is looked up. The resulting value is a built-in primitive `IO` action. The next step is to execute this `IO` action, resulting in a value that represents the standard input stream, which has type `IO.FS.Stream`. Standard input is then associated with the name to the left of the arrow (here `stdin`) for the remainder of the `do` block.

Executing the second line, `let stdout ← IO.getStdout`, proceeds similarly. First, the expression `IO.getStdout` is evaluated, yielding an `IO` action that will return the standard output. Next, this action is executed, actually returning the standard output. Finally, this value is associated with the name `stdout` for the remainder of the `do` block.

Asking a Question

Now that `stdin` and `stdout` have been found, the remainder of the block consists of a question and an answer:

```
  stdout.putStrLn "How would you like to be addressed?"
  let input ← stdin.getLine
  let name := input.dropRightWhile Char.isWhitespace
  stdout.putStrLn s!"Hello, {name}!"
```

The first statement in the block, `stdout.putStrLn "How would you like to be addressed?"`, consists of an expression. To execute an expression, it is first evaluated. In this case, `IO.FS.Stream.putStrLn` has type `IO.FS.Stream → String → IO Unit`. This means that it is a function that accepts a stream and a string, returning an `IO` action. The expression uses accessor notation for a function call. This function is applied to two arguments: the standard output stream and a string. The value of the expression is an `IO` action that will write the string and a newline character to the output stream. Having found this value, the next step is to execute it, which causes the string and newline to actually be written to `stdout`. Statements that consist only of expressions do not introduce any new variables.

The next statement in the block is `let input ← stdin.getLine`. `IO.FS.Stream.getLine` has type `IO.FS.Stream → IO String`, which means that it is a function from a stream to an `IO` action that will return a string. Once again, this is an example of accessor notation. This `IO` action is executed, and the program waits until the user has typed a complete line of input. Assume the user writes "`David`". The resulting line (`"David\n"`) is associated with `input`, where the escape sequence `\n` denotes the newline character.

```
  let name := input.dropRightWhile Char.isWhitespace
  stdout.putStrLn s!"Hello, {name}!"
```

The next line, `let name := input.dropRightWhile Char.isWhitespace`, is a `let` statement. Unlike the other `let` statements in this program, it uses `:=` instead of `←`. This means that the expression will be evaluated, but the resulting value need not be an `IO` action and will not be executed. In this case, `String.dropRightWhile` takes a string and a predicate over characters and returns a new string from which all the characters at the end of the string that satisfy the predicate have been removed. For example,

```
#eval "Hello!!!".dropRightWhile (· == '!')
```

yields

```
"Hello"
```

and

```
#eval "Hello...   ".dropRightWhile (fun c => not (c.isAlphanum))
```

yields

```
"Hello"
```

in which all non-alphanumeric characters have been removed from the right side of the string. In the current line of the program, whitespace characters (including the newline) are removed from the right side of the input string, resulting in `"David"`, which is associated with `name` for the remainder of the block.

Greeting the User

All that remains to be executed in the `do` block is a single statement:

```
  stdout.putStrLn s!"Hello, {name}!"
```

The string argument to `putStrLn` is constructed via string interpolation, yielding the string `"Hello, David!"`. Because this statement is an expression, it is evaluated to yield an `IO` action that will print this string with a newline to standard output. Once the expression has been evaluated, the resulting `IO` action is executed, resulting in the greeting.

`IO` Actions as Values

In the above description, it can be difficult to see why the distinction between evaluating expressions and executing `IO` actions is necessary. After all, each action is executed immediately after it is produced. Why not simply carry out the effects during evaluation, as is done in other languages?

The answer is twofold. First off, separating evaluation from execution means that programs must be explicit about which functions can have side effects. Because the parts of the program that do not have effects are much more amenable to mathematical reasoning, whether in the heads of programmers or using Lean's facilities for formal proof, this separation can make it easier to avoid bugs. Secondly, not all `IO` actions need be executed at the time that they come into existence. The ability to mention an action without carrying it out allows ordinary functions to be used as control structures.

For example, the function `twice` takes an `IO` action as its argument, returning a new action that will execute the argument action twice.

```
def twice (action : IO Unit) : IO Unit := do
  action
  action
```

Executing

```
twice (IO.println "shy")
```

results in

```
shy
shy
```

being printed. This can be generalized to a version that runs the underlying action any number of times:

```
def nTimes (action : IO Unit) : Nat → IO Unit
  | 0 => pure ()
  | n + 1 => do
    action
    nTimes action n
```

In the base case for `Nat.zero`, the result is `pure ()`. The function `pure` creates an `IO` action that has no side effects, but returns `pure`'s argument, which in this case is the constructor for `Unit`. As an action that does nothing and returns nothing interesting, `pure ()` is at the same time utterly boring and very useful. In the recursive step, a `do` block is used to create an action that first executes `action` and then executes the result of the recursive call. Executing `nTimes (IO.println "Hello") 3` causes the following output:

```
Hello
Hello
Hello
```

In addition to using functions as control structures, the fact that `IO` actions are first-class values means that they can be saved in data structures for later execution. For instance, the function `countdown` takes a `Nat` and returns a list of unexecuted `IO` actions, one for each `Nat`:

```
def countdown : Nat → List (IO Unit)
  | 0 => [IO.println "Blast off!"]
  | n + 1 => IO.println s!"{n + 1}" :: countdown n
```

This function has no side effects, and does not print anything. For example, it can be applied to an argument, and the length of the resulting list of actions can be checked:

```
def from5 : List (IO Unit) := countdown 5
```

This list contains six elements (one for each number, plus a `"Blast off!"` action for zero):

```
#eval from5.length
```

```
6
```

The function `runActions` takes a list of actions and constructs a single action that runs them all in order:

```
def runActions : List (IO Unit) → IO Unit
  | [] => pure ()
  | act :: actions => do
    act
    runActions actions
```

Its structure is essentially the same as that of `nTimes`, except instead of having one action that is executed for each `Nat.succ`, the action under each `List.cons` is to be executed. Similarly, `runActions` does not itself run the actions. It creates a new action that will run them, and that action must be placed in a position where it will be executed as a part of `main`:

```
def main : IO Unit := runActions from5
```

Running this program results in the following output:

```
5
4
3
2
1
Blast off!
```

What happens when this program is run? The first step is to evaluate `main`. That occurs as follows:

```
main
===>
runActions from5
===>
runActions (countdown 5)
===>
runActions
  [IO.println "5",
   IO.println "4",
   IO.println "3",
   IO.println "2",
   IO.println "1",
   IO.println "Blast off!"]
===>
do IO.println "5"
   IO.println "4"
   IO.println "3"
   IO.println "2"
   IO.println "1"
   IO.println "Blast off!"
   pure ()
```

The resulting `IO` action is a `do` block. Each step of the `do` block is then executed, one at a time, yielding the expected output. The final step, `pure ()`, does not have any effects, and it is only present because the definition of `runActions` needs a base case.

Exercise

Step through the execution of the following program on a piece of paper:

```
def main : IO Unit := do
  let englishGreeting := IO.println "Hello!"
  IO.println "Bonjour!"
  englishGreeting
```

While stepping through the program's execution, identify when an expression is being evaluated and when an `IO` action is being executed. When executing an `IO` action results in a side effect, write it down. After doing this, run the program with Lean and double-check that your predictions about the side effects were correct.

Starting a Project

As a program written in Lean becomes more serious, an ahead-of-time compiler-based workflow that results in an executable becomes more attractive. Like other languages, Lean has tools for building multiple-file packages and managing dependencies. The standard Lean build tool is called Lake (short for "Lean Make"). Lake is typically configured using a TOML file that declaratively specifies dependencies and describes what is to be built. For advanced use cases, Lake can also be configured in Lean itself.

First steps

To get started with a project that uses Lake, use the command `lake new greeting` in a directory that does not already contain a file or directory called `greeting`. This creates a directory called `greeting` that contains the following files:

- `Main.lean` is the file in which the Lean compiler will look for the `main` action.
- `Greeting.lean` and `Greeting/Basic.lean` are the scaffolding of a support library for the program.
- `lakefile.toml` contains the configuration that `lake` needs to build the application.
- `lean-toolchain` contains an identifier for the specific version of Lean that is used for the project.

Additionally, `lake new` initializes the project as a Git repository and configures its `.gitignore` file to ignore intermediate build products. Typically, the majority of the application logic will be in a collection of libraries for the program, while `Main.lean` will contain a small wrapper around these pieces that does things like parsing command lines and executing the central application logic. To create a project in an already-existing directory, run `lake init` instead of `lake new`.

By default, the library file `Greeting/Basic.lean` contains a single definition:

```
def hello := "world"
```

The library file `Greeting.lean` imports `Greeting/Basic.lean`:

```
-- This module serves as the root of the `Greeting` library.
-- Import modules here that should be built as part of the library.
import Greeting.Basic
```

This means that everything defined in `Greetings/Basic.lean` is also available to files that import `Greetings.lean`. In `import` statements, dots are interpreted as directories on disk.

The executable source `Main.lean` contains:

```
import Greeting

def main : IO Unit :=
  IO.println s!"Hello, {hello}!"
```

Because `Main.lean` imports `Greetings.lean` and `Greetings.lean` imports `Greetings/Basic.lean`, the definition of `hello` is available in `main`.

To build the package, run the command `lake build`. After a number of build commands scroll by, the resulting binary has been placed in `.lake/build/bin`. Running `./.lake/build/bin/greeting` results in `Hello, world!`. Instead of running the binary directly, the command `lake exe` can be used to build the binary if necessary and then run it. Running `lake exe greeting` also results in `Hello, world!`.

Lakefiles

A `lakefile.lean` describes a *package*, which is a coherent collection of Lean code for distribution, analogous to an `npm` or `nuget` package or a Rust crate. A package may contain any number of libraries or executables. The documentation for Lake describes the available options in a Lake configuration. The generated `lakefile.toml` contains the following:

```
name = "greeting"
version = "0.1.0"
defaultTargets = ["greeting"]

[[lean_lib]]
name = "Greeting"

[[lean_exe]]
name = "greeting"
root = "Main"
```

This initial Lake configuration consists of three items:

- *package* settings, at the top of the file,
- a *library* declaration, named `Greeting`, and
- an *executable*, named `greeting`.

Each Lake configuration file will contain exactly one package, but any number of dependencies, libraries, or executables. By convention, package and executable names begin with a lowercase letter, while libraries begin with an uppercase letter. Dependencies are declarations of other Lean packages (either locally or from remote Git repositories) The items in the Lake configuration file allow things like source file locations, module hierarchies, and compiler flags to be configured. Generally speaking, however, the defaults are reasonable. Lake configuration files written in the Lean format may additionally contain *external libraries*, which are libraries not written in Lean to be statically linked with the resulting executable, *custom targets*, which are build targets that don't fit naturally into the library/executable taxonomy, and *scripts*, which are essentially `IO` actions (similar to `main`), but that additionally have access to metadata about the package configuration.

Libraries, executables, and custom targets are all called *targets*. By default, `lake build` builds those targets that are specified in the `defaultTargets` list. To build a target that is not a default target, specify the target's name as an argument after `lake build`.

Libraries and Imports

A Lean library consists of a hierarchically organized collection of source files from which names can be imported, called *modules*. By default, a library has a single root file that matches its name. In this case, the root file for the library `Greeting` is `Greeting.lean`. The first line of `Main.lean`, which is `import Greeting`, makes the contents of `Greeting.lean` available in `Main.lean`.

Additional module files may be added to the library by creating a directory called `Greeting` and placing them inside. These names can be imported by replacing the directory separator with a dot. For instance, creating the file `Greeting/Smile.lean` with the contents:

```
def Expression.happy : String := "a big smile"
```

means that `Main.lean` can use the definition as follows:

```
import Greeting
import Greeting.Smile

open Expression

def main : IO Unit :=
  IO.println s!"Hello, {hello}, with {happy}!"
```

The module name hierarchy is decoupled from the namespace hierarchy. In Lean, modules are units of code distribution, while namespaces are units of code organization. That is, names defined in the module `Greeting.Smile` are not automatically in a corresponding namespace `Greeting.Smile`. In particular, `happy` is in the `Expression` namespace. Modules may place names into any namespace they like, and the code that imports them may `open` the namespace or not. `import` is used to make the contents of a source file available, while `open` makes names from a namespace available in the current context without prefixes.

The line `open Expression` makes the name `Expression.happy` accessible as `happy` in `main`. Namespaces may also be opened *selectively*, making only some of their names available without explicit prefixes. This is done by writing the desired names in parentheses. For example, `Nat.toFloat` converts a natural number to a `Float`. It can be made available as `toFloat` using `open Nat (toFloat)`.

Worked Example: `cat`

The standard Unix utility `cat` takes a number of command-line options, followed by zero or more input files. If no files are provided, or if one of them is a dash (`-`), then it takes the standard input as the corresponding input instead of reading a file. The contents of the inputs are written, one after the other, to the standard output. If a specified input file does not exist, this is noted on standard error, but `cat` continues concatenating the remaining inputs. A non-zero exit code is returned if any of the input files do not exist.

This section describes a simplified version of `cat`, called `feline`. Unlike commonly-used versions of `cat`, `feline` has no command-line options for features such as numbering lines, indicating non-printing characters, or displaying help text. Furthermore, it cannot read more than once from a standard input that's associated with a terminal device.

To get the most benefit from this section, follow along yourself. It's OK to copy-paste the code examples, but it's even better to type them in by hand. This makes it easier to learn the mechanical process of typing in code, recovering from mistakes, and interpreting feedback from the compiler.

Getting started

The first step in implementing `feline` is to create a package and decide how to organize the code. In this case, because the program is so simple, all the code will be placed in `Main.lean`. The first step is to run `lake new feline`. Edit the Lakefile to remove the library, and delete the generated library code and the reference to it from `Main.lean`. Once this has been done, `lakefile.lean` should contain:

```
name = "feline"
version = "0.1.0"
defaultTargets = ["feline"]

[[lean_exe]]
name = "feline"
root = "Main"
```

and `Main.lean` should contain something like:

```
def main : IO Unit :=
  IO.println s!"Hello, cats!"
```

Alternatively, running `lake new feline exe` instructs `lake` to use a template that does not include a library section, making it unnecessary to edit the file.

Ensure that the code can be built by running `lake build`.

Concatenating Streams

Now that the basic skeleton of the program has been built, it's time to actually enter the code. A proper implementation of `cat` can be used with infinite IO streams, such as `/dev/random`, which means that it can't read its input into memory before outputting it. Furthermore, it should not work one character at a time, as this leads to frustratingly slow performance. Instead, it's better to read contiguous blocks of data all at once, directing the data to the standard output one block at a time.

The first step is to decide how big of a block to read. For the sake of simplicity, this implementation uses a conservative 20 kilobyte block. `USize` is analogous to `size_t` in C—it's an unsigned integer type that is big enough to represent all valid array sizes.

```
def bufsize : USize := 20 * 1024
```

Streams

The main work of `feline` is done by `dump`, which reads input one block at a time, dumping the result to standard output, until the end of the input has been reached. The end of the input is indicated by `read` returning an empty byte array:

```
partial def dump (stream : IO.FS.Stream) : IO Unit := do
  let buf ← stream.read bufsize
  if buf.isEmpty then
    pure ()
  else
    let stdout ← IO.getStdout
    stdout.write buf
    dump stream
```

The `dump` function is declared `partial`, because it calls itself recursively on input that is not immediately smaller than an argument. When a function is declared to be partial, Lean does not require a proof that it terminates. On the other hand, partial functions are also much less amenable to proofs of correctness, because allowing infinite loops in Lean's logic would make it unsound. However, there is no way to prove that `dump` terminates, because infinite input (such as from `/dev/random`) would mean that it does not, in fact, terminate. In cases like this, there is no alternative to declaring the function `partial`.

The type `IO.FS.Stream` represents a POSIX stream. Behind the scenes, it is represented as a structure that has one field for each POSIX stream operation. Each operation is represented as an IO action that provides the corresponding operation:

```
structure Stream where
  flush   : IO Unit
  read    : USize → IO ByteArray
  write   : ByteArray → IO Unit
  getLine : IO String
  putStr  : String → IO Unit
  isTty   : BaseIO Bool
```

The type `BaseIO` is a variant of `IO` that rules out run-time errors. The Lean compiler contains `IO` actions (such as `IO.getStdout`, which is called in `dump`) to get streams that represent standard input, standard output, and standard error. These are `IO` actions rather than ordinary definitions because Lean allows these standard POSIX streams to be replaced in a process, which makes it easier to do things like capturing the output from a program into a string by writing a custom `IO.FS.Stream`.

The control flow in `dump` is essentially a `while` loop. When `dump` is called, if the stream has reached the end of the file, `pure ()` terminates the function by returning the constructor for `Unit`. If the stream has not yet reached the end of the file, one block is read, and its contents are written to `stdout`, after which `dump` calls itself directly. The recursive calls continue until `stream.read` returns an empty byte array, which indicates the end of the file.

When an `if` expression occurs as a statement in a `do`, as in `dump`, each branch of the `if` is implicitly provided with a `do`. In other words, the sequence of steps following the `else` are treated as a sequence of `IO` actions to be executed, just as if they had a `do` at the beginning. Names introduced with `let` in the branches of the `if` are visible only in their own branches, and are not in scope outside of the `if`.

There is no danger of running out of stack space while calling `dump` because the recursive call happens as the very last step in the function, and its result is returned directly rather than being manipulated or computed with. This kind of recursion is called *tail recursion*, and it is described in more detail later in this book. Because the compiled code does not need to retain any state, the Lean compiler can compile the recursive call to a jump.

If `feline` only redirected standard input to standard output, then `dump` would be sufficient. However, it also needs to be able to open files that are provided as command-line arguments and emit their contents. When its argument is the name of a file that exists, `fileStream` returns a stream that reads the file's contents. When the argument is not a file, `fileStream` emits an error and returns `none`.

```
def fileStream (filename : System.FilePath) : IO (Option IO.FS.Stream) := do
  let fileExists ← filename.pathExists
  if not fileExists then
    let stderr ← IO.getStderr
    stderr.putStrLn s!"File not found: {filename}"
    pure none
  else
    let handle ← IO.FS.Handle.mk filename IO.FS.Mode.read
    pure (some (IO.FS.Stream.ofHandle handle))
```

Opening a file as a stream takes two steps. First, a file handle is created by opening the file in read mode. A Lean file handle tracks an underlying file descriptor. When there are no references to the file handle value, a finalizer closes the file descriptor. Second, the file handle is given the same interface as a POSIX stream using `IO.FS.Stream.ofHandle`, which fills each field of the `Stream` structure with the corresponding `IO` action that works on file handles.

Handling Input

The main loop of `feline` is another tail-recursive function, called `process`. In order to return a non-zero exit code if any of the inputs could not be read, `process` takes an argument `exitCode` that represents the current exit code for the whole program. Additionally, it takes a list of input files to be processed.

```
def process (exitCode : UInt32) (args : List String) : IO UInt32 := do
  match args with
  | [] => pure exitCode
  | "-" :: args =>
    let stdin ← IO.getStdin
    dump stdin
    process exitCode args
  | filename :: args =>
    let stream ← fileStream ⟨filename⟩
    match stream with
    | none =>
      process 1 args
    | some stream =>
      dump stream
      process exitCode args
```

Just as with `if`, each branch of a `match` that is used as a statement in a `do` is implicitly provided with its own `do`.

There are three possibilities. One is that no more files remain to be processed, in which case `process` returns the error code unchanged. Another is that the specified filename is `"-"`, in which case `process` dumps the contents of the standard input and then processes the remaining filenames. The final possibility is that an actual filename was specified. In this case, `fileStream` is used to attempt to open the file as a POSIX stream. Its argument is encased in `⟨ ... ⟩` because a `FilePath` is a single-field structure that contains a string. If the file could not be opened, it is skipped, and the recursive call to `process` sets the exit code to `1`. If it could, then it is dumped, and the recursive call to `process` leaves the exit code unchanged.

`process` does not need to be marked `partial` because it is structurally recursive. Each recursive call is provided with the tail of the input list, and all Lean lists are finite. Thus, `process` does not introduce any non-termination.

Main

The final step is to write the `main` action. Unlike prior examples, `main` in `feline` is a function. In Lean, `main` can have one of three types:

- `main : IO Unit` corresponds to programs that cannot read their command-line arguments and always indicate success with an exit code of `0`,
- `main : IO UInt32` corresponds to `int main(void)` in C, for programs without arguments that return exit codes, and
- `main : List String → IO UInt32` corresponds to `int main(int argc, char **argv)` in C, for programs that take arguments and signal success or failure.

If no arguments were provided, `feline` should read from standard input as if it were called with a single `"-"` argument. Otherwise, the arguments should be processed one after the other.

```
def main (args : List String) : IO UInt32 :=
  match args with
  | [] => process 0 ["-"]
  | _ =>  process 0 args
```

Meow!

To check whether `feline` works, the first step is to build it with `lake build`. First off, when called without arguments, it should emit what it receives from standard input. Check that

```
echo "It works!" | lake exe feline
```

emits `It works!`.

Secondly, when called with files as arguments, it should print them. If the file `test1.txt` contains

```
It's time to find a warm spot
```

and `test2.txt` contains

```
and curl up!
```

then the command

```
lake exe feline test1.txt test2.txt
```

should emit

```
It's time to find a warm spot
and curl up!
```

Finally, the `-` argument should be handled appropriately.

```
echo "and purr" | lake exe feline test1.txt - test2.txt
```

should yield

```
It's time to find a warm spot
and purr
and curl up!
```

Exercise

Extend `feline` with support for usage information. The extended version should accept a command-line argument `--help` that causes documentation about the available command-line options to be written to standard output.

Additional Conveniences

Nested Actions

Many of the functions in `feline` exhibit a repetitive pattern in which an `IO` action's result is given a name, and then used immediately and only once. For instance, in `dump`:

```
partial def dump (stream : IO.FS.Stream) : IO Unit := do
  let buf ← stream.read bufsize
  if buf.isEmpty then
    pure ()
  else
    let stdout ← IO.getStdout
    stdout.write buf
    dump stream
```

the pattern occurs for `stdout`:

```
    let stdout ← IO.getStdout
    stdout.write buf
```

Similarly, `fileStream` contains the following snippet:

```
  let fileExists ← filename.pathExists
  if not fileExists then
```

When Lean is compiling a `do` block, expressions that consist of a left arrow immediately under parentheses are lifted to the nearest enclosing `do`, and their results are bound to a unique name. This unique name replaces the origin of the expression. This means that `dump` can also be written as follows:

```
partial def dump (stream : IO.FS.Stream) : IO Unit := do
  let buf ← stream.read bufsize
  if buf.isEmpty then
    pure ()
  else
    (← IO.getStdout).write buf
    dump stream
```

This version of `dump` avoids introducing names that are used only once, which can greatly simplify a program. `IO` actions that Lean lifts from a nested expression context are called *nested actions*.

`fileStream` can be simplified using the same technique:

```
def fileStream (filename : System.FilePath) : IO (Option IO.FS.Stream) := do
  if not (← filename.pathExists) then
    (← IO.getStderr).putStrLn s!"File not found: {filename}"
    pure none
  else
    let handle ← IO.FS.Handle.mk filename IO.FS.Mode.read
    pure (some (IO.FS.Stream.ofHandle handle))
```

In this case, the local name of `handle` could also have been eliminated using nested actions, but the resulting expression would have been long and complicated. Even though it's often good style to use nested actions, it can still sometimes be helpful to name intermediate results.

It is important to remember, however, that nested actions are only a shorter notation for `IO` actions that occur in a surrounding `do` block. The side effects that are involved in executing them still occur in the same order, and execution of side effects is not interspersed with the evaluation of expressions. Therefore, nested actions cannot be lifted from the branches of an `if`.

For an example of where this might be confusing, consider the following helper definitions that return data after announcing to the world that they have been executed:

```
def getNumA : IO Nat := do
  (← IO.getStdout).putStrLn "A"
  pure 5

def getNumB : IO Nat := do
  (← IO.getStdout).putStrLn "B"
  pure 7
```

These definitions are intended to stand in for more complicated `IO` code that might validate user input, read a database, or open a file.

A program that prints `0` when number A is five, or number `B` otherwise, might be written as follows:

```
def test : IO Unit := do
  let a : Nat := if (← getNumA) == 5 then 0 else (← getNumB)
  (← IO.getStdout).putStrLn s!"The answer is {a}"
```

This program would be equivalent to:

```
def test : IO Unit := do
  let x ← getNumA
  let y ← getNumB
  let a : Nat := if x == 5 then 0 else y
  (← IO.getStdout).putStrLn s!"The answer is {a}"
```

which runs `getNumB` regardless of whether the result of `getNumA` is equal to `5`. To prevent this confusion, nested actions are not allowed in an `if` that is not itself a line in the `do`, and the following error message results:

```
invalid use of `(<- ...)`, must be nested inside a 'do' expression
```

Flexible Layouts for do

In Lean, `do` expressions are whitespace-sensitive. Each `IO` action or local binding in the `do` is expected to start on its own line, and they should all have the same indentation. Almost all uses of `do` should be written this way. In some rare contexts, however, manual control over whitespace and indentation may be necessary, or it may be convenient to have multiple small actions on a single line. In these cases, newlines can be replaced with a semicolon and indentation can be replaced with curly braces.

For instance, all of the following programs are equivalent:

```
-- This version uses only whitespace-sensitive layout
def main : IO Unit := do
  let stdin ← IO.getStdin
  let stdout ← IO.getStdout

  stdout.putStrLn "How would you like to be addressed?"
  let name := (← stdin.getLine).trim
  stdout.putStrLn s!"Hello, {name}!"

-- This version is as explicit as possible
def main : IO Unit := do {
  let stdin ← IO.getStdin;
  let stdout ← IO.getStdout;

  stdout.putStrLn "How would you like to be addressed?";
  let name := (← stdin.getLine).trim;
  stdout.putStrLn s!"Hello, {name}!"
}

-- This version uses a semicolon to put two actions on the same line
def main : IO Unit := do
  let stdin ← IO.getStdin; let stdout ← IO.getStdout

  stdout.putStrLn "How would you like to be addressed?"
  let name := (← stdin.getLine).trim
  stdout.putStrLn s!"Hello, {name}!"
```

Idiomatic Lean code uses curly braces with `do` very rarely.

Running `IO` Actions With `#eval`

Lean's `#eval` command can be used to execute `IO` actions, rather than just evaluating them. Normally, adding a `#eval` command to a Lean file causes Lean to evaluate the provided expression, convert the resulting value to a string, and provide that string as a tooltip and in the info window. Rather than failing because `IO` actions can't be converted to strings, `#eval` executes them, carrying out their side effects. If the result of execution is the `Unit` value `()`, then no result string is shown, but if it is a type that can be converted to a string, then Lean displays the resulting value.

This means that, given the prior definitions of `countdown` and `runActions`,

```
#eval runActions (countdown 3)
```

displays

```
3
2
1
Blast off!
```

This is the output produced by running the `IO` action, rather than some opaque representation of the action itself. In other words, for `IO` actions, `#eval` both *evaluates* the provided expression and *executes* the resulting action value.

Quickly testing `IO` actions with `#eval` can be much more convenient that compiling and running whole programs. However, there are some limitations. For instance, reading from standard input simply returns empty input. Additionally, the `IO` action is re-executed whenever Lean needs to update the diagnostic information that it provides to users, and this can happen at unpredictable times. An action that reads and writes files, for instance, may do so unexpectedly.

Summary

Evaluation vs Execution

Side effects are aspects of program execution that go beyond the evaluation of mathematical expressions, such as reading files, throwing exceptions, or triggering industrial machinery. While most languages allow side effects to occur during evaluation, Lean does not. Instead, Lean has a type called `IO` that represents *descriptions* of programs that use side effects. These descriptions are then executed by the language's run-time system, which invokes the Lean expression evaluator to carry out specific computations. Values of type `IO α` are called *`IO` actions*. The simplest is `pure`, which returns its argument and has no actual side effects.

`IO` actions can also be understood as functions that take the whole world as an argument and return a new world in which the side effect has occurred. Behind the scenes, the `IO` library ensures that the world is never duplicated, created, or destroyed. While this model of side effects cannot actually be implemented, as the whole universe is too big to fit in memory, the real world can be represented by a token that is passed around through the program.

An `IO` action `main` is executed when the program starts. `main` can have one of three types:

- `main : IO Unit` is used for simple programs that cannot read their command-line arguments and always return exit code `0`,
- `main : IO UInt32` is used for programs without arguments that may signal success or failure, and
- `main : List String → IO UInt32` is used for programs that take command-line arguments and signal success or failure.

`do` Notation

The Lean standard library provides a number of basic `IO` actions that represent effects such as reading from and writing to files and interacting with standard input and standard output. These base `IO` actions are composed into larger `IO` actions using `do` notation, which is a built-in domain-specific language for writing descriptions of programs with side effects. A `do` expression contains a sequence of *statements*, which may be:

- expressions that represent `IO` actions,
- ordinary local definitions with `let` and `:=`, where the defined name refers to the value of the provided expression, or
- local definitions with `let` and `←`, where the defined name refers to the result of executing the value of the provided expression.

`IO` actions that are written with `do` are executed one statement at a time.

Furthermore, `if` and `match` expressions that occur immediately under a `do` are implicitly considered to have their own `do` in each branch. Inside of a `do` expression, *nested actions* are expressions with a left arrow immediately under parentheses. The Lean compiler implicitly lifts them to the nearest enclosing `do`, which may be implicitly part of a branch of a `match` or `if` expression, and gives them a unique name. This unique name then replaces the origin site of the nested action.

Compiling and Running Programs

A Lean program that consists of a single file with a `main` definition can be run using `lean --run FILE`. While this can be a nice way to get started with a simple program, most programs will eventually graduate to a multiple-file project that should be compiled before running.

Lean projects are organized into *packages*, which are collections of libraries and executables together with information about dependencies and a build configuration. Packages are described using Lake, a Lean build tool. Use `lake new` to create a Lake package in a new directory, or `lake init` to create one in the current directory. Lake package configuration is another domain-specific language. Use `lake build` to build a project.

Partiality

One consequence of following the mathematical model of expression evaluation is that every expression must have a value. This rules out both incomplete pattern matches that fail to cover all constructors of a datatype and programs that can fall into an infinite loop. Lean ensures that all `match` expressions cover all cases, and that all recursive functions are either structurally recursive or have an explicit proof of termination.

However, some real programs require the possibility of looping infinitely, because they handle potentially-infinite data, such as POSIX streams. Lean provides an escape hatch:

functions whose definition is marked `partial` are not required to terminate. This comes at a cost. Because types are a first-class part of the Lean language, functions can return types. Partial functions, however, are not evaluated during type checking, because an infinite loop in a function could cause the type checker to enter an infinite loop. Furthermore, mathematical proofs are unable to inspect the definitions of partial functions, which means that programs that use them are much less amenable to formal proof.

Interlude: Propositions, Proofs, and Indexing

Like many languages, Lean uses square brackets for indexing into arrays and lists. For instance, if `woodlandCritters` is defined as follows:

```
def woodlandCritters : List String :=
  ["hedgehog", "deer", "snail"]
```

then the individual components can be extracted:

```
def hedgehog := woodlandCritters[0]
def deer := woodlandCritters[1]
def snail := woodlandCritters[2]
```

However, attempting to extract the fourth element results in a compile-time error, rather than a run-time error:

```
def oops := woodlandCritters[3]
```

```
failed to prove index is valid, possible solutions:
  - Use `have`-expressions to prove the index is valid
  - Use `a[i]!` notation instead, runtime check is performed, and 'Panic' error
message is produced if index is not valid
  - Use `a[i]?` notation instead, result is an `Option` type
  - Use `a[i]'h` notation instead, where `h` is a proof that index is valid
⊢ 3 < woodlandCritters.length
```

This error message is saying Lean tried to automatically mathematically prove that `3 < woodlandCritters.length` (i.e. `3 < List.length woodlandCritters`), which would mean that the lookup was safe, but that it could not do so. Out-of-bounds errors are a common class of bugs, and Lean uses its dual nature as a programming language and a theorem prover to rule out as many as possible.

Understanding how this works requires an understanding of three key ideas: propositions, proofs, and tactics.

Propositions and Proofs

A *proposition* is a statement that can be true or false. All of the following English sentences are propositions:

- 1 + 1 = 2
- Addition is commutative.
- There are infinitely many prime numbers.
- 1 + 1 = 15
- Paris is the capital of France.
- Buenos Aires is the capital of South Korea.
- All birds can fly.

On the other hand, nonsense statements are not propositions. Despite being grammatical, none of the following are propositions:

- 1 + green = ice cream
- All capital cities are prime numbers.
- At least one gorg is a fleep.

Propositions come in two varieties: those that are purely mathematical, relying only on our definitions of concepts, and those that are facts about the world. Theorem provers like Lean are concerned with the former category, and have nothing to say about the flight capabilities of penguins or the legal status of cities.

A *proof* is a convincing argument that a proposition is true. For mathematical propositions, these arguments make use of the definitions of the concepts that are involved as well as the rules of logical argumentation. Most proofs are written for people to understand, and leave out many tedious details. Computer-aided theorem provers like Lean are designed to allow mathematicians to write proofs while omitting many details, and it is the software's responsibility to fill in the missing explicit steps. These steps can be mechanically checked. This decreases the likelihood of oversights or mistakes.

In Lean, a program's type describes the ways it can be interacted with. For instance, a program of type `Nat → List String` is a function that takes a `Nat` argument and produces a list of strings. In other words, each type specifies what counts as a program with that type.

In Lean, propositions are in fact types. They specify what counts as evidence that the statement is true. The proposition is proved by providing this evidence, which is checked by Lean. On the other hand, if the proposition is false, then it will be impossible to construct this evidence.

For example, the proposition "1 + 1 = 2" can be written directly in Lean. The evidence for this proposition is the constructor `rfl`, which is short for *reflexivity*. In mathematics, a relation is *reflexive* if every element is related to itself; this is a basic requirement in order to have a sensible notion of equality. Because `1 + 1` computes to `2`, they are really the same thing:

```
def onePlusOneIsTwo : 1 + 1 = 2 := rfl
```

On the other hand, `rfl` does not prove the false proposition "1 + 1 = 15":

```
def onePlusOneIsFifteen : 1 + 1 = 15 := rfl
```

```
type mismatch
  rfl
has type
  ?m.911 = ?m.911 : Prop
but is expected to have type
  1 + 1 = 15 : Prop
```

This error message indicates that `rfl` can prove that two expressions are equal when both sides of the equality statement are already the same number. Because `1 + 1` evaluates directly to `2`, they are considered to be the same, which allows `onePlusOneIsTwo` to be accepted. Just as `Type` describes types such as `Nat`, `String`, and `List (Nat × String × (Int → Float))` that represent data structures and functions, `Prop` describes propositions.

When a proposition has been proven, it is called a *theorem*. In Lean, it is conventional to declare theorems with the `theorem` keyword instead of `def`. This helps readers see which declarations are intended to be read as mathematical proofs, and which are definitions. Generally speaking, with a proof, what matters is that there is evidence that a proposition is true, but it's not particularly important *which* evidence was provided. With definitions, on the other hand, it matters very much which particular value is selected—after all, a definition of addition that always returns `0` is clearly wrong. Because the details of a proof don't matter for later proofs, using the `theorem` keyword enables greater parallelism in the Lean compiler.

The prior example could be rewritten as follows:

```
def OnePlusOneIsTwo : Prop := 1 + 1 = 2

theorem onePlusOneIsTwo : OnePlusOneIsTwo := rfl
```

Tactics

Proofs are normally written using *tactics*, rather than by providing evidence directly. Tactics are small programs that construct evidence for a proposition. These programs run in a *proof state* that tracks the statement that is to be proved (called the *goal*) along with the assumptions that are available to prove it. Running a tactic on a goal results in a new proof state that contains new goals. The proof is complete when all goals have been proven.

To write a proof with tactics, begin the definition with `by`. Writing `by` puts Lean into tactic mode until the end of the next indented block. While in tactic mode, Lean provides ongoing feedback about the current proof state. Written with tactics, `onePlusOneIsTwo` is still quite short:

```
theorem onePlusOneIsTwo : 1 + 1 = 2 := by
  decide
```

The `decide` tactic invokes a *decision procedure*, which is a program that can check whether a statement is true or false, returning a suitable proof in either case. It is primarily used when working with concrete values like `1` and `2`. The other important tactic in this book is `simp`, short for "simplify", which is the workhorse of Lean proofs. It rewrites the goal to as simple a form as possible. In many cases, this rewriting simplifies the statement so much that it can be automatically proved. Behind the scenes, a detailed formal proof is constructed, but using `simp` hides this complexity.

Tactics are useful for a number of reasons:

1. Many proofs are complicated and tedious when written out down to the smallest detail, and tactics can automate these uninteresting parts.
2. Proofs written with tactics are easier to maintain over time, because flexible automation can paper over small changes to definitions.
3. Because a single tactic can prove many different theorems, Lean can use tactics behind the scenes to free users from writing proofs by hand. For instance, an array lookup requires a proof that the index is in bounds, and a tactic can typically construct that proof without the user needing to worry about it.

Behind the scenes, indexing notation uses a tactic to prove that the user's lookup operation is safe. This tactic takes many facts about arithemtic into account, combining them with any locally-known facts to attempt to prove that the index is in bounds.

Connectives

The basic building blocks of logic, such as "and", "or", "true", "false", and "not", are called *logical connectives*. Each connective defines what counts as evidence of its truth. For example, to prove a statement "*A* and *B*", one must prove both *A* and *B*. This means that evidence for "*A* and *B*" is a pair that contains both evidence for *A* and evidence for *B*. Similarly, evidence for "*A* or *B*" consists of either evidence for *A* or evidence for *B*.

In particular, most of these connectives are defined like datatypes, and they have constructors. If `A` and `B` are propositions, then "`A` and `B`" (written `A ∧ B`) is a proposition. Evidence for `A ∧ B` consists of the constructor `And.intro`, which has the type `A → B → A ∧ B`. Replacing `A` and `B` with concrete propositions, it is possible to prove `1 + 1 = 2 ∧ "Str".append "ing" = "String"` with `And.intro rfl rfl`. Of course, `decide` is also powerful enough to find this proof:

```
theorem addAndAppend : 1 + 1 = 2 ∧ "Str".append "ing" = "String" := by decide
```

Similarly, "`A` or `B`" (written `A ∨ B`) has two constructors, because a proof of "`A` or `B`" requires only that one of the two underlying propositions be true. There are two constructors: `Or.inl`, with type `A → A ∨ B`, and `Or.inr`, with type `B → A ∨ B`.

Implication (if *A* then *B*) is represented using functions. In particular, a function that transforms evidence for *A* into evidence for *B* is itself evidence that *A* implies *B*. This is different from the usual description of implication, in which `A → B` is shorthand for `¬A ∨ B`, but the two formulations are equivalent.

Because evidence for an "and" is a constructor, it can be used with pattern matching. For instance, a proof that *A* and *B* implies *A* or *B* is a function that pulls the evidence of *A* (or of *B*) out of the evidence for *A* and *B*, and then uses this evidence to produce evidence of *A* or *B*:

```
theorem andImpliesOr : A ∧ B → A ∨ B :=
  fun andEvidence =>
    match andEvidence with
    | And.intro a b => Or.inl a
```

Connective	Lean Syntax	Evidence
True	`True`	`True.intro : True`
False	`False`	No evidence
A and *B*	`A ∧ B`	`And.intro : A → B → A ∧ B`
A or *B*	`A ∨ B`	Either `Or.inl : A → A ∨ B` or `Or.inr : B → A ∨ B`
A implies *B*	`A → B`	A function that transforms evidence of *A* into evidence of *B*
not *A*	`¬A`	A function that would transform evidence of *A* into evidence of `False`

The `decide` tactic can prove theorems that use these connectives. For example:

```
theorem onePlusOneAndLessThan : 1 + 1 = 2 ∨ 3 < 5 := by decide
theorem notTwoEqualFive : ¬(1 + 1 = 5) := by decide
theorem trueIsTrue : True := by decide
theorem trueOrFalse : True ∨ False := by decide
theorem falseImpliesTrue : False → True := by decide
```

Evidence as Arguments

In some cases, safely indexing into a list requires that the list have some minimum size, but the list itself is a variable rather than a concrete value. For this lookup to be safe, there must be some evidence that the list is long enough. One of the easiest ways to make indexing safe is to have the function that performs a lookup into a data structure take the required evidence of safety as an argument. For instance, a function that returns the third entry in a list is not generally safe because lists might contain zero, one, or two entries:

```
def third (xs : List α) : α := xs[2]
```

```
failed to prove index is valid, possible solutions:
  - Use `have`-expressions to prove the index is valid
  - Use `a[i]!` notation instead, runtime check is performed, and 'Panic' error
message is produced if index is not valid
  - Use `a[i]?` notation instead, result is an `Option` type
  - Use `a[i]'h` notation instead, where `h` is a proof that index is valid
α : Type ?u.4181
xs : List α
⊢ 2 < xs.length
```

However, the obligation to show that the list has at least three entries can be imposed on the caller by adding an argument that consists of evidence that the indexing operation is safe:

```
def third (xs : List α) (ok : xs.length > 2) : α := xs[2]
```

In this example, `xs.length > 2` is not a program that checks *whether* `xs` has more than 2 entries. It is a proposition that could be true or false, and the argument `ok` must be evidence that it is true.

When the function is called on a concrete list, its length is known. In these cases, `by decide` can construct the evidence automatically:

```
#eval third woodlandCritters (by decide)
```

```
"snail"
```

Indexing Without Evidence

In cases where it's not practical to prove that an indexing operation is in bounds, there are other alternatives. Adding a question mark results in an `Option`, where the result is `some` if the index is in bounds, and `none` otherwise. For example:

```
def thirdOption (xs : List α) : Option α := xs[2]?
```

```
#eval thirdOption woodlandCritters
```

```
some "snail"
```

```
#eval thirdOption ["only", "two"]
```

```
none
```

There is also a version that crashes the program when the index is out of bounds, rather than returning an `Option`:

```
#eval woodlandCritters[1]!
```

```
"deer"
```

Messages You May Meet

In addition to proving that a statement is true, the `decide` tactic can also prove that it is false. When asked to prove that a one-element list has more than two elements, it returns an error that indicates that the statement is indeed false:

```
#eval third ["rabbit"] (by decide)
```

```
tactic 'decide' proved that the proposition
  ["rabbit"].length > 2
is false
```

The `simp` and `decide` tactics do not automatically unfold definitions with `def`. Attempting to prove `OnePlusOneIsTwo` using `simp` fails:

```
theorem onePlusOneIsStillTwo : OnePlusOneIsTwo := by simp
```

The error messages simply states that it could do nothing, because without unfolding `OnePlusOneIsTwo`, no progress can be made:

```
simp made no progress
```

Using `decide` also fails:

```
theorem onePlusOneIsStillTwo : OnePlusOneIsTwo := by decide
```

This is also due to it not unfolding `OnePlusOneIsTwo`:

```
failed to synthesize
  Decidable OnePlusOneIsTwo

Additional diagnostic information may be available using the `set_option
diagnostics true` command.
```

Defining `OnePlusOneIsTwo` with `abbrev` fixes the problem by marking the definition for unfolding.

In addition to the error that occurs when Lean is unable to find compile-time evidence that an indexing operation is safe, polymorphic functions that use unsafe indexing may produce the following message:

```
def unsafeThird (xs : List α) : α := xs[2]!
```

```
failed to synthesize
  Inhabited α

Additional diagnostic information may be available using the `set_option
diagnostics true` command.
```

This is due to a technical restriction that is part of keeping Lean usable as both a logic for proving theorems and a programming language. In particular, only programs whose types contain at least one value are allowed to crash. This is because a proposition in Lean is a kind of type that classifies evidence of its truth. False propositions have no such evidence. If a program with an empty type could crash, then that crashing program could be used as a kind of fake evidence for a false proposition.

Internally, Lean contains a table of types that are known to have at least one value. This error is saying that some arbitrary type `α` is not necessarily in that table. The next chapter describes how to add to this table, and how to successfully write functions like `unsafeThird`.

Adding whitespace between a list and the brackets used for lookup can cause another message:

```
#eval woodlandCritters [1]
```

```
function expected at
  woodlandCritters
term has type
  List String
```

Adding a space causes Lean to treat the expression as a function application, and the index as a list that contains a single number. This error message results from having Lean attempt to treat `woodlandCritters` as a function.

Exercises

- Prove the following theorems using `rfl`: `2 + 3 = 5`, `15 - 8 = 7`, `"Hello, ".append "world" = "Hello, world"`. What happens if `rfl` is used to prove `5 < 18`? Why?
- Prove the following theorems using `by decide`: `2 + 3 = 5`, `15 - 8 = 7`, `"Hello, ".append "world" = "Hello, world"`, `5 < 18`.
- Write a function that looks up the fifth entry in a list. Pass the evidence that this lookup is safe as an argument to the function.

Overloading and Type Classes

In many languages, the built-in datatypes get special treatment. For example, in C and Java, `+` can be used to add `float`s and `int`s, but not arbitrary-precision numbers from a third-party library. Similarly, numeric literals can be used directly for the built-in types, but not for user-defined number types. Other languages provide an *overloading* mechanism for operators, where the same operator can be given a meaning for a new type. In these languages, such as C++ and C#, a wide variety of built-in operators can be overloaded, and the compiler uses the type checker to select a particular implementation.

In addition to numeric literals and operators, many languages allow overloading of functions or methods. In C++, Java, C# and Kotlin, multiple implementations of a method are allowed, with differing numbers and types of arguments. The compiler uses the number of arguments and their types to determine which overload was intended.

Function and operator overloading has a key limitation: polymorphic functions can't restrict their type arguments to types for which a given overload exists. For example, an overloaded method might be defined for strings, byte arrays, and file pointers, but there's no way to write a second method that works for any of these. Instead, this second method must itself be overloaded for each type that has an overload of the original method, resulting in many boilerplate definitions instead of a single polymorphic definition. Another consequence of this restriction is that some operators (such as equality in Java) end up being defined for *every* combination of arguments, even when it is not necessarily sensible to do so. If programmers are not very careful, this can lead to programs that crash at runtime or silently compute an incorrect result.

Lean implements overloading using a mechanism called *type classes*, pioneered in Haskell, that allows overloading of operators, functions, and literals in a manner that works well with polymorphism. A type class describes a collection of overloadable operations. To overload these operations for a new type, an *instance* is created that contains an implementation of each operation for the new type. For example, a type class named `Add` describes types that allow addition, and an instance of `Add` for `Nat` provides an implementation of addition for `Nat`.

The terms *class* and *instance* can be confusing for those who are used to object-oriented languages, because they are not closely related to classes and instances in object-oriented languages. However, they do share common roots: in everyday language, the term "class" refers to a group that shares some common attributes. While classes in object-oriented programming certainly describe groups of objects with common attributes, the term additionally refers to a specific mechanism in a programming language for describing such a group. Type classes are also a means of describing types that share common attributes (namely, implementations of certain operations), but they don't really have anything else in common with classes as found in object-oriented programming.

A Lean type class is much more analogous to a Java or C# *interface*. Both type classes and interfaces describe a conceptually related set of operations that are implemented for a type or collection of types. Similarly, an instance of a type class is akin to the code in a Java or C# class that is prescribed by the implemented interfaces, rather than an instance of a Java or C# class. Unlike Java or C#'s interfaces, types can be given instances for type classes that the author of the type does not have access to. In this way, they are very similar to Rust traits.

Positive Numbers

In some applications, only positive numbers make sense. For example, compilers and interpreters typically use one-indexed line and column numbers for source positions, and a datatype that represents only non-empty lists will never report a length of zero. Rather than relying on natural numbers, and littering the code with assertions that the number is not zero, it can be useful to design a datatype that represents only positive numbers.

One way to represent positive numbers is very similar to `Nat`, except with `one` as the base case instead of `zero`:

```
inductive Pos : Type where
  | one : Pos
  | succ : Pos → Pos
```

This datatype represents exactly the intended set of values, but it is not very convenient to use. For example, numeric literals are rejected:

```
def seven : Pos := 7
```

```
failed to synthesize
  OfNat Pos 7
numerals are polymorphic in Lean, but the numeral `7` cannot be used in a
context where the expected type is
  Pos
due to the absence of the instance above

Additional diagnostic information may be available using the `set_option
diagnostics true` command.
```

Instead, the constructors must be used directly:

```
def seven : Pos :=
  Pos.succ (Pos.succ (Pos.succ (Pos.succ (Pos.succ (Pos.succ Pos.one)))))
```

Similarly, addition and multiplication are not easy to use:

```
def fourteen : Pos := seven + seven
```

```
failed to synthesize
  HAdd Pos Pos ?m.332

Additional diagnostic information may be available using the `set_option
diagnostics true` command.
```

```
def fortyNine : Pos := seven * seven
```

```
failed to synthesize
  HMul Pos Pos ?m.332

Additional diagnostic information may be available using the `set_option
diagnostics true` command.
```

Each of these error messages begins with `failed to synthesize instance`. This indicates that the error is due to an overloaded operation that has not been implemented, and it describes the type class that must be implemented.

Classes and Instances

A type class consists of a name, some parameters, and a collection of *methods*. The parameters describe the types for which overloadable operations are being defined, and the methods are the names and type signatures of the overloadable operations. Once again, there is a terminology clash with object-oriented languages. In object-oriented programming, a method is essentially a function that is connected to a particular object in memory, with special access to the object's private state. Objects are interacted with via their methods. In Lean, the term "method" refers to an operation that has been declared to be overloadable, with no special connection to objects or values or private fields.

One way to overload addition is to define a type class named `Plus`, with an addition method named `plus`. Once an instance of `Plus` for `Nat` has been defined, it becomes possible to add two `Nat`s using `Plus.plus`:

```
#eval Plus.plus 5 3
```

```
8
```

Adding more instances allows `Plus.plus` to take more types of arguments.

In the following type class declaration, `Plus` is the name of the class, `α : Type` is the only argument, and `plus : α → α → α` is the only method:

```
class Plus (α : Type) where
  plus : α → α → α
```

This declaration says that there is a type class `Plus` that overloads operations with respect to a type `α`. In particular, there is one overloaded operation called `plus` that takes two `α`s and returns an `α`.

Type classes are first class, just as types are first class. In particular, a type class is another kind of type. The type of `Plus` is `Type → Type`, because it takes a type as an argument (`α`) and results in a new type that describes the overloading of `Plus`'s operation for `α`.

To overload `plus` for a particular type, write an instance:

```
instance : Plus Nat where
  plus := Nat.add
```

The colon after `instance` indicates that `Plus Nat` is indeed a type. Each method of class `Plus` should be assigned a value using `:=`. In this case, there is only one method: `plus`.

By default, type class methods are defined in a namespace with the same name as the type class. It can be convenient to `open` the namespace so that users don't need to type the name of the class first. Parentheses in an `open` command indicate that only the indicated names from the namespace are to be made accessible:

```
open Plus (plus)

#eval plus 5 3
```

```
8
```

Defining an addition function for `Pos` and an instance of `Plus Pos` allows `plus` to be used to add both `Pos` and `Nat` values:

```
def Pos.plus : Pos → Pos → Pos
  | Pos.one, k => Pos.succ k
  | Pos.succ n, k => Pos.succ (n.plus k)

instance : Plus Pos where
  plus := Pos.plus

def fourteen : Pos := plus seven seven
```

Because there is not yet an instance of `Plus Float`, attempting to add two floating-point numbers with `plus` fails with a familiar message:

```
#eval plus 5.2 917.25861
```

```
failed to synthesize
  Plus Float

Additional diagnostic information may be available using the `set_option
diagnostics true` command.
```

These errors mean that Lean was unable to find an instance for a given type class.

Overloaded Addition

Lean's built-in addition operator is syntactic sugar for a type class called `HAdd`, which flexibly allows the arguments to addition to have different types. `HAdd` is short for *heterogeneous addition*. For example, an `HAdd` instance can be written to allow a `Nat` to be added to a `Float`, resulting in a new `Float`. When a programmer writes `x + y`, it is interpreted as meaning `HAdd.hAdd x y`.

While an understanding of the full generality of `HAdd` relies on features that are discussed in another section in this chapter, there is a simpler type class called `Add` that does not allow the types of the arguments to be mixed. The Lean libraries are set up so that an instance of `Add` will be found when searching for an instance of `HAdd` in which both arguments have the same type.

Defining an instance of `Add Pos` allows `Pos` values to use ordinary addition syntax:

```
instance : Add Pos where
  add := Pos.plus

def fourteen : Pos := seven + seven
```

Conversion to Strings

Another useful built-in class is called `ToString`. Instances of `ToString` provide a standard way of converting values from a given type into strings. For example, a `ToString` instance is used when a value occurs in an interpolated string, and it determines how the `IO.println` function used at the beginning of the description of `IO` will display a value.

For example, one way to convert a `Pos` into a `String` is to reveal its inner structure. The function `posToString` takes a `Bool` that determines whether to parenthesize uses of `Pos.succ`, which should be `true` in the initial call to the function and `false` in all recursive calls.

```
def posToString (atTop : Bool) (p : Pos) : String :=
  let paren s := if atTop then s else "(" ++ s ++ ")"
  match p with
  | Pos.one => "Pos.one"
  | Pos.succ n => paren s!"Pos.succ {posToString false n}"
```

Using this function for a `ToString` instance:

```
instance : ToString Pos where
  toString := posToString true
```

results in informative, yet overwhelming, output:

```
#eval s!"There are {seven}"
```

```
"There are Pos.succ (Pos.succ (Pos.succ (Pos.succ (Pos.succ (Pos.succ
Pos.one)))))"
```

On the other hand, every positive number has a corresponding `Nat`. Converting it to a `Nat` and then using the `ToString Nat` instance (that is, the overloading of `toString` for `Nat`) is a quick way to generate much shorter output:

```
def Pos.toNat : Pos → Nat
  | Pos.one => 1
  | Pos.succ n => n.toNat + 1

instance : ToString Pos where
  toString x := toString (x.toNat)

#eval s!"There are {seven}"
```

```
"There are 7"
```

When more than one instance is defined, the most recent takes precedence. Additionally, if a type has a `ToString` instance, then it can be used to display the result of `#eval` even if the type in question was not defined with `deriving Repr`, so `#eval seven` outputs `7`.

Overloaded Multiplication

For multiplication, there is a type class called `HMul` that allows mixed argument types, just like `HAdd`. Just as `x + y` is interpreted as `HAdd.hAdd x y`, `x * y` is interpreted as `HMul.hMul x y`. For the common case of multiplication of two arguments with the same type, a `Mul` instance suffices.

An instance of `Mul` allows ordinary multiplication syntax to be used with `Pos`:

```
def Pos.mul : Pos → Pos → Pos
  | Pos.one, k => k
  | Pos.succ n, k => n.mul k + k

instance : Mul Pos where
  mul := Pos.mul
```

With this instance, multiplication works as expected:

```
#eval [seven * Pos.one,
       seven * seven,
       Pos.succ Pos.one * seven]
```

```
[7, 49, 14]
```

Literal Numbers

It is quite inconvenient to write out a sequence of constructors for positive numbers. One way to work around the problem would be to provide a function to convert a `Nat` into a `Pos`. However, this approach has downsides. First off, because `Pos` cannot represent `0`, the resulting function would either convert a `Nat` to a bigger number, or it would return `Option Pos`. Neither is particularly convenient for users. Secondly, the need to call the function explicitly would make programs that use positive numbers much less convenient to write than programs that use `Nat`. Having a trade-off between precise types and convenient APIs means that the precise types become less useful.

There are two type classes that are used to overload numeric literals: `Zero` and `OfNat`. Because many types have values that are naturally written with `0`, the `Zero` class allow these specific values to be overridden. It is defined as follows:

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

Because `0` is not a positive number, there should be no instance of `Zero Pos`.

In Lean, natural number literals are interpreted using a type class called `OfNat`:

```
class OfNat (α : Type) (_ : Nat) where
  ofNat : α
```

This type class takes two arguments: `α` is the type for which a natural number is overloaded, and the unnamed `Nat` argument is the actual literal number that was encountered in the program. The method `ofNat` is then used as the value of the numeric literal. Because the class contains the `Nat` argument, it becomes possible to define only instances for those values where the number makes sense.

`OfNat` demonstrates that the arguments to type classes do not need to be types. Because types in Lean are first-class participants in the language that can be passed as arguments to functions and given definitions with `def` and `abbrev`, there is no barrier that prevents non-type arguments in positions where a less-flexible language could not permit them. This flexibility allows overloaded operations to be provided for particular values as well as particular types. Additionally, it allows the Lean standard library to arrange for there to be a `Zero α` instance whenever there's an `OfNat α 0` instance, and vice versa.

A sum type that represents natural numbers less than four can be defined as follows:

```
inductive LT4 where
  | zero
  | one
  | two
  | three
deriving Repr
```

While it would not make sense to allow *any* literal number to be used for this type, numbers less than four clearly make sense:

```
instance : OfNat LT4 0 where
  ofNat := LT4.zero

instance : OfNat LT4 1 where
  ofNat := LT4.one

instance : OfNat LT4 2 where
  ofNat := LT4.two

instance : OfNat LT4 3 where
  ofNat := LT4.three
```

With these instances, the following examples work:

```
#eval (3 : LT4)
```

```
LT4.three
```

```
#eval (0 : LT4)
```

```
LT4.zero
```

On the other hand, out-of-bounds literals are still not allowed:

```
#eval (4 : LT4)
```

```
failed to synthesize
  OfNat LT4 4
numerals are polymorphic in Lean, but the numeral `4` cannot be used in a
context where the expected type is
  LT4
due to the absence of the instance above

Additional diagnostic information may be available using the `set_option
diagnostics true` command.
```

For `Pos`, the `OfNat` instance should work for *any* `Nat` other than `Nat.zero`. Another way to phrase this is to say that for all natural numbers `n`, the instance should work for `n + 1`. Just as names like `α` automatically become implicit arguments to functions that Lean fills out on its own, instances can take automatic implicit arguments. In this instance, the argument `n` stands for any `Nat`, and the instance is defined for a `Nat` that's one greater:

```
instance : OfNat Pos (n + 1) where
  ofNat :=
    let rec natPlusOne : Nat → Pos
      | 0 => Pos.one
      | k + 1 => Pos.succ (natPlusOne k)
    natPlusOne n
```

Because `n` stands for a `Nat` that's one less than what the user wrote, the helper function `natPlusOne` returns a `Pos` that's one greater than its argument. This makes it possible to use natural number literals for positive numbers, but not for zero:

```
def eight : Pos := 8
```

```
def zero : Pos := 0
```

```
failed to synthesize
  OfNat Pos 0
numerals are polymorphic in Lean, but the numeral `0` cannot be used in a
context where the expected type is
  Pos
due to the absence of the instance above

Additional diagnostic information may be available using the `set_option
diagnostics true` command.
```

Exercises

Another Representation

An alternative way to represent a positive number is as the successor of some `Nat`. Replace the definition of `Pos` with a structure whose constructor is named `succ` that contains a `Nat`:

```
structure Pos where
  succ ::
  pred : Nat
```

Define instances of `Add`, `Mul`, `ToString`, and `OfNat` that allow this version of `Pos` to be used conveniently.

Even Numbers

Define a datatype that represents only even numbers. Define instances of `Add`, `Mul`, and `ToString` that allow it to be used conveniently. `OfNat` requires a feature that is introduced in the next section.

HTTP Requests

An HTTP request begins with an identification of a HTTP method, such as `GET` or `POST`, along with a URI and an HTTP version. Define an inductive type that represents an interesting subset of the HTTP methods, and a structure that represents HTTP responses. Responses should have a `ToString` instance that makes it possible to debug them. Use a type class to associate different `IO` actions with each HTTP method, and write a test harness as an `IO` action that calls each method and prints the result.

Type Classes and Polymorphism

It can be useful to write functions that work for *any* overloading of a given function. For instance, `IO.println` works for any type that has an instance of `ToString`. This is indicated using square brackets around the required instance: the type of `IO.println` is `{α : Type} → [ToString α] → α → IO Unit`. This type says that `IO.println` accepts an argument of type `α`, which Lean should determine automatically, and that there must be a `ToString` instance available for `α`. It returns an `IO` action.

Checking Polymorphic Functions' Types

Checking the type of a function that takes implicit arguments or uses type classes requires the use of some additional syntax. Simply writing

```
#check (IO.println)
```

yields a type with metavariables:

```
IO.println : ?m.2620 → IO Unit
```

This is because Lean does its best to discover implicit arguments, and the presence of metavariables indicates that it did not yet discover enough type information to do so. To understand the signature of a function, this feature can be suppressed with an at-sign (`@`) before the function's name:

```
#check @IO.println
```

```
@IO.println : {α : Type u_1} → [inst : ToString α] → α → IO Unit
```

In this output, the instance itself has been given the name `inst`. Additionally, there is a `u_1` after `Type`, which uses a feature of Lean that has not yet been introduced. For now, ignore these parameters to `Type`.

Defining Polymorphic Functions with Instance Implicits

A function that sums all entries in a list needs two instances: `Add` allows the entries to be added, and an `OfNat` instance for `0` provides a sensible value to return for the empty list:

```
def List.sumOfContents [Add α] [OfNat α 0] : List α → α
  | [] => 0
  | x :: xs => x + xs.sumOfContents
```

This function can be also defined with a `Zero α` requirement instead of `OfNat α 0`. Both are equivalent, but `Zero α` can be easier to read:

```
def List.sumOfContents [Add α] [Zero α] : List α → α
  | [] => 0
  | x :: xs => x + xs.sumOfContents
```

This function can be used for a list of `Nat`s:

```
def fourNats : List Nat := [1, 2, 3, 4]

#eval fourNats.sumOfContents
```

```
10
```

but not for a list of `Pos` numbers:

```
def fourPos : List Pos := [1, 2, 3, 4]

#eval fourPos.sumOfContents
```

```
failed to synthesize
  Zero Pos

Additional diagnostic information may be available using the `set_option
diagnostics true` command.
```

The Lean standard library includes this function, where it is called `List.sum`.

Specifications of required instances in square brackets are called *instance implicits*. Behind the scenes, every type class defines a structure that has a field for each overloaded operation. Instances are values of that structure type, with each field containing an implementation. At a call site, Lean is responsible for finding an instance value to pass for each instance implicit argument. The most important difference between ordinary implicit arguments and instance implicits is the strategy that Lean uses to find an argument value. In the case of ordinary implicit arguments, Lean uses a technique called *unification* to find a single unique argument value that would allow the program to pass the type checker. This process relies only on the specific types involved in the function's definition and the call site.

For instance implicits, Lean instead consults a built-in table of instance values.

Just as the `OfNat` instance for `Pos` took a natural number `n` as an automatic implicit argument, instances may also take instance implicit arguments themselves. The section on polymorphism presented a polymorphic point type:

```
structure PPoint (α : Type) where
  x : α
  y : α
deriving Repr
```

Addition of points should add the underlying `x` and `y` fields. Thus, an `Add` instance for `PPoint` requires an `Add` instance for whatever type these fields have. In other words, the `Add` instance for `PPoint` requires a further `Add` instance for `α`:

```
instance [Add α] : Add (PPoint α) where
  add p1 p2 := { x := p1.x + p2.x, y := p1.y + p2.y }
```

When Lean encounters an addition of two points, it searches for and finds this instance. It then performs a further search for the `Add α` instance.

The instance values that are constructed in this way are values of the type class's structure type. A successful recursive instance search results in a structure value that has a reference to another structure value. An instance of `Add (PPoint Nat)` contains a reference to the instance of `Add Nat` that was found.

This recursive search process means that type classes offer significantly more power than plain overloaded functions. A library of polymorphic instances is a set of code building blocks that the compiler will assemble on its own, given nothing but the desired type. Polymorphic functions that take instance arguments are latent requests to the type class mechanism to assemble helper functions behind the scenes. The API's clients are freed from the burden of plumbing together all of the necessary parts by hand.

Methods and Implicit Arguments

The type of `@OfNat.ofNat` may be surprising. It is `{α : Type} → (n : Nat) → [OfNat α n] → α`, in which the `Nat` argument `n` occurs as an explicit function parameter. In the declaration of the method, however, `ofNat` simply has type `α`. This seeming discrepancy is because declaring a type class really results in the following:

- A structure type to contain the implementation of each overloaded operation

- A namespace with the same name as the class
- For each method, a function in the class's namespace that retrieves its implementation from an instance

This is analogous to the way that declaring a new structure also declares accessor functions. The primary difference is that a structure's accessors take the structure value as an explicit parameter, while the type class methods take the instance value as an instance implicit to be found automatically by Lean.

In order for Lean to find an instance, its parameters must be available. This means that each parameter to the type class must be a parameter to the method that occurs before the instance. It is most convenient when these parameters are implicit, because Lean does the work of discovering their values. For example, `@Add.add` has the type `{α : Type} → [Add α] → α → α → α`. In this case, the type parameter `α` can be implicit because the arguments to `Add.add` provide information about which type the user intended. This type can then be used to search for the `Add` instance.

In the case of `ofNat`, however, the particular `Nat` literal to be decoded does not appear as part of any other parameter's type. This means that Lean would have no information to use when attempting to figure out the implicit parameter `n`. The result would be a very inconvenient API. Thus, in these cases, Lean uses an explicit parameter for the class's method.

Exercises

Even Number Literals

Write an instance of `OfNat` for the even number datatype from the previous section's exercises that uses recursive instance search.

Recursive Instance Search Depth

There is a limit to how many times the Lean compiler will attempt a recursive instance search. This places a limit on the size of even number literals defined in the previous exercise. Experimentally determine what the limit is.

Controlling Instance Search

An instance of the `Add` class is sufficient to allow two expressions with type `Pos` to be conveniently added, producing another `Pos`. However, in many cases, it can be useful to be more flexible and allow *heterogeneous* operator overloading, where the arguments may have different types. For example, adding a `Nat` to a `Pos` or a `Pos` to a `Nat` will always yield a `Pos`:

```
def addNatPos : Nat → Pos → Pos
  | 0, p => p
  | n + 1, p => Pos.succ (addNatPos n p)

def addPosNat : Pos → Nat → Pos
  | p, 0 => p
  | p, n + 1 => Pos.succ (addPosNat p n)
```

These functions allow natural numbers to be added to positive numbers, but they cannot be used with the `Add` type class, which expects both arguments to `add` to have the same type.

Heterogeneous Overloadings

As mentioned in the section on overloaded addition, Lean provides a type class called `HAdd` for overloading addition heterogeneously. The `HAdd` class takes three type parameters: the two argument types and the return type. Instances of `HAdd Nat Pos Pos` and `HAdd Pos Nat Pos` allow ordinary addition notation to be used to mix the types:

```
instance : HAdd Nat Pos Pos where
  hAdd := addNatPos

instance : HAdd Pos Nat Pos where
  hAdd := addPosNat
```

Given the above two instances, the following examples work:

```
#eval (3 : Pos) + (5 : Nat)
```

```
8
```

```
#eval (3 : Nat) + (5 : Pos)
```

```
8
```

The definition of the `HAdd` type class is very much like the following definition of `HPlus` with the corresponding instances:

```
class HPlus (α : Type) (β : Type) (γ : Type) where
  hPlus : α → β → γ

instance : HPlus Nat Pos Pos where
  hPlus := addNatPos

instance : HPlus Pos Nat Pos where
  hPlus := addPosNat
```

However, instances of `HPlus` are significantly less useful than instances of `HAdd`. When attempting to use these instances with `#eval`, an error occurs:

```
-- TODO CommandM snuck in here, find a new example!
#eval HPlus.hPlus (3 : Pos) (5 : Nat)
```

```
failed to synthesize
  HPlus Pos Nat (Lean.Elab.Command.CommandElabM ?α)

Additional diagnostic information may be available using the `set_option
diagnostics true` command.
```

This happens because there is a metavariable in the type, and Lean has no way to solve it.

As discussed in the initial description of polymorphism, metavariables represent unknown parts of a program that could not be inferred. When an expression is written following `#eval`, Lean attempts to determine its type automatically. In this case, it could not. Because the third type parameter for `HPlus` was unknown, Lean couldn't carry out type class instance search, but instance search is the only way that Lean could determine the expression's type. That is, the `HPlus Pos Nat Pos` instance can only apply if the expression should have type `Pos`, but there's nothing in the program other than the instance itself to indicate that it should have this type.

One solution to the problem is to ensure that all three types are available by adding a type annotation to the whole expression:

```
#eval (HPlus.hPlus (3 : Pos) (5 : Nat) : Pos)
```

```
8
```

However, this solution is not very convenient for users of the positive number library.

Output Parameters

This problem can also be solved by declaring `γ` to be an *output parameter*. Most type class parameters are inputs to the search algorithm: they are used to select an instance. For example, in an `OfNat` instance, both the type and the natural number are used to select a particular interpretation of a natural number literal. However, in some cases, it can be convenient to start the search process even when some of the type parameters are not yet known, and use the instances that are discovered in the search to determine values for metavariables. The parameters that aren't needed to start instance search are outputs of the process, which is declared with the `outParam` modifier:

```
class HPlus (α : Type) (β : Type) (γ : outParam Type) where
  hPlus : α → β → γ
```

With this output parameter, type class instance search is able to select an instance without knowing `γ` in advance. For instance:

```
#eval HPlus.hPlus (3 : Pos) (5 : Nat)
```

```
8
```

It might be helpful to think of output parameters as defining a kind of function. Any given instance of a type class that has one or more output parameters provides Lean with instructions for determining the outputs from the inputs. The process of searching for an instance, possibly recursively, ends up being more powerful than mere overloading. Output parameters can determine other types in the program, and instance search can assemble a collection of underlying instances into a program that has this type.

Default Instances

Deciding whether a parameter is an input or an output controls the circumstances under which Lean will initiate type class search. In particular, type class search does not occur until all inputs are known. However, in some cases, output parameters are not enough, and instance search should also occur when some inputs are unknown. This is a bit like default values for optional function arguments in Python or Kotlin, except default *types* are being selected.

Default instances are instances that are available for instance search *even when not all their inputs are known*. When one of these instances can be used, it will be used. This can cause programs to successfully type check, rather than failing with errors related to unknown types and metavariables. On the other hand, default instances can make instance selection less predictable. In particular, if an undesired default instance is selected, then an expression may have a different type than expected, which can cause confusing type errors to occur elsewhere in the program. Be selective about where default instances are used!

One example of where default instances can be useful is an instance of `HPlus` that can be derived from an `Add` instance. In other words, ordinary addition is a special case of heterogeneous addition in which all three types happen to be the same. This can be implemented using the following instance:

```
instance [Add α] : HPlus α α α where
  hPlus := Add.add
```

With this instance, `hPlus` can be used for any addable type, like `Nat`:

```
#eval HPlus.hPlus (3 : Nat) (5 : Nat)
```

```
8
```

However, this instance will only be used in situations where the types of both arguments are known. For example,

```
#check HPlus.hPlus (5 : Nat) (3 : Nat)
```

yields the type

```
HPlus.hPlus 5 3 : Nat
```

as expected, but

```
#check HPlus.hPlus (5 : Nat)
```

yields a type that contains two metavariables, one for the remaining argument and one for the return type:

```
HPlus.hPlus 5 : ?m.6076 → ?m.6078
```

In the vast majority of cases, when someone supplies one argument to addition, the other argument will have the same type. To make this instance into a default instance, apply the `default_instance` attribute:

```
@[default_instance]
instance [Add α] : HPlus α α α where
  hPlus := Add.add
```

With this default instance, the example has a more useful type:

```
#check HPlus.hPlus (5 : Nat)
```

yields

```
HPlus.hPlus 5 : Nat → Nat
```

Each operator that exists in overloadable heterogeneous and homogeneous versions follows the pattern of a default instance that allows the homogeneous version to be used in contexts where the heterogeneous is expected. The infix operator is replaced with a call to the heterogeneous version, and the homogeneous default instance is selected when possible.

Similarly, simply writing `5` gives a `Nat` rather than a type with a metavariable that is waiting for more information in order to select an `OfNat` instance. This is because the `OfNat` instance for `Nat` is a default instance.

Default instances can also be assigned *priorities* that affect which will be chosen in situations where more than one might apply. For more information on default instance priorities, please consult the Lean manual.

Exercises

Define an instance of `HMul (PPoint α) α (PPoint α)` that multiplies both projections by the scalar. It should work for any type `α` for which there is a `Mul α` instance. For example,

```
#eval {x := 2.5, y := 3.7 : PPoint Float} * 2.0
```

should yield

```
{ x := 5.000000, y := 7.400000 }
```

Arrays and Indexing

The Interlude describes how to use indexing notation in order to look up entries in a list by their position. This syntax is also governed by a type class, and it can be used for a variety of different types.

Arrays

For instance, Lean arrays are much more efficient than linked lists for most purposes. In Lean, the type `Array α` is a dynamically-sized array holding values of type `α`, much like a Java `ArrayList`, a C++ `std::vector`, or a Rust `Vec`. Unlike `List`, which has a pointer indirection on each use of the `cons` constructor, arrays occupy a contiguous region of memory, which is much better for processor caches. Also, looking up a value in an array takes constant time, while lookup in a linked list takes time proportional to the index being accessed.

In pure functional languages like Lean, it is not possible to mutate a given position in a data structure. Instead, a copy is made that has the desired modifications. However, copying is not always necessary: the Lean compiler and runtime contain an optimization that can allow modifications to be implemented as mutations behind the scenes when there is only a single unique reference to an array.

Arrays are written similarly to lists, but with a leading `#`:

```
def northernTrees : Array String :=
  #["sloe", "birch", "elm", "oak"]
```

The number of values in an array can be found using `Array.size`. For instance, `northernTrees.size` evaluates to `4`. For indices that are smaller than an array's size, indexing notation can be used to find the corresponding value, just as with lists. That is, `northernTrees[2]` evaluates to `"elm"`. Similarly, the compiler requires a proof that an index is in bounds, and attempting to look up a value outside the bounds of the array results in a compile-time error, just as with lists. For instance, `northernTrees[8]` results in:

```
failed to prove index is valid, possible solutions:
  - Use `have`-expressions to prove the index is valid
  - Use `a[i]!` notation instead, runtime check is performed, and 'Panic' error
message is produced if index is not valid
  - Use `a[i]?` notation instead, result is an `Option` type
  - Use `a[i]'h` notation instead, where `h` is a proof that index is valid
⊢ 8 < northernTrees.size
```

Non-Empty Lists

A datatype that represents non-empty lists can be defined as a structure with a field for the head of the list and a field for the tail, which is an ordinary, potentially empty list:

```
structure NonEmptyList (α : Type) : Type where
  head : α
  tail : List α
```

For example, the non-empty list `idahoSpiders` (which contains some spider species native to the US state of Idaho) consists of `"Banded Garden Spider"` followed by four other spiders, for a total of five spiders:

```
def idahoSpiders : NonEmptyList String := {
  head := "Banded Garden Spider",
  tail := [
    "Long-legged Sac Spider",
    "Wolf Spider",
    "Hobo Spider",
    "Cat-faced Spider"
  ]
}
```

Looking up the value at a specific index in this list with a recursive function should consider three possibilities:

1. The index is `0`, in which case the head of the list should be returned.
2. The index is `n + 1` and the tail is empty, in which case the index is out of bounds.
3. The index is `n + 1` and the tail is non-empty, in which case the function can be called recursively on the tail and `n`.

For example, a lookup function that returns an `Option` can be written as follows:

```
def NonEmptyList.get? : NonEmptyList α → Nat → Option α
  | xs, 0 => some xs.head
  | {head := _, tail := []}, _ + 1 => none
  | {head := _, tail := h :: t}, n + 1 => get? {head := h, tail := t} n
```

Each case in the pattern match corresponds to one of the possibilities above. The recursive call to `get?` does not require a `NonEmptyList` namespace qualifier because the body of the definition is implicitly in the definition's namespace. Another way to write this function uses `get?` for lists when the index is greater than zero:

```
def NonEmptyList.get? : NonEmptyList α → Nat → Option α
  | xs, 0 => some xs.head
  | xs, n + 1 => xs.tail[n]?
```

If the list contains one entry, then only `0` is a valid index. If it contains two entries, then both `0` and `1` are valid indices. If it contains three entries, then `0`, `1`, and `2` are valid indices. In other words, the valid indices into a non-empty list are natural numbers that are strictly less than the length of the list, which are less than or equal to the length of the tail.

The definition of what it means for an index to be in bounds should be written as an `abbrev` because the tactics used to find evidence that indices are acceptable are able to solve inequalities of numbers, but they don't know anything about the name `NonEmptyList.inBounds`:

```
abbrev NonEmptyList.inBounds (xs : NonEmptyList α) (i : Nat) : Prop :=
  i ≤ xs.tail.length
```

This function returns a proposition that might be true or false. For instance, `2` is in bounds for `idahoSpiders`, while `5` is not:

```
theorem atLeastThreeSpiders : idahoSpiders.inBounds 2 := by decide

theorem notSixSpiders : ¬idahoSpiders.inBounds 5 := by decide
```

The logical negation operator has a very low precedence, which means that `¬idahoSpiders.inBounds 5` is equivalent to `¬(idahoSpiders.inBounds 5)`.

This fact can be used to write a lookup function that requires evidence that the index is valid, and thus need not return `Option`, by delegating to the version for lists that checks the evidence at compile time:

```
def NonEmptyList.get (xs : NonEmptyList α) (i : Nat) (ok : xs.inBounds i) : α :=
  match i with
  | 0 => xs.head
  | n + 1 => xs.tail[n]
```

It is, of course, possible to write this function to use the evidence directly, rather than delegating to a standard library function that happens to be able to use the same evidence. This requires techniques for working with proofs and propositions that are described later in this book.

Overloading Indexing

Indexing notation for a collection type can be overloaded by defining an instance of the `GetElem` type class. For the sake of flexiblity, `GetElem` has four parameters:

- The type of the collection
- The type of the index
- The type of elements that are extracted from the collection
- A function that determines what counts as evidence that the index is in bounds

The element type and the evidence function are both output parameters. `GetElem` has a single method, `getElem`, which takes a collection value, an index value, and evidence that the index is in bounds as arguments, and returns an element:

```
class GetElem (coll : Type) (idx : Type) (item : outParam Type) (inBounds :
outParam (coll → idx → Prop)) where
  getElem : (c : coll) → (i : idx) → inBounds c i → item
```

In the case of `NonEmptyList α`, these parameters are:

- The collection is `NonEmptyList α`
- Indices have type `Nat`
- The type of elements is `α`
- An index is in bounds if it is less than or equal to the length of the tail

In fact, the `GetElem` instance can delegate directly to `NonEmptyList.get`:

```
instance : GetElem (NonEmptyList α) Nat α NonEmptyList.inBounds where
  getElem := NonEmptyList.get
```

With this instance, `NonEmptyList` becomes just as convenient to use as `List`. Evaluating `idahoSpiders[0]` yields `"Banded Garden Spider"`, while `idahoSpiders[9]` leads to the compile-time error:

```
failed to prove index is valid, possible solutions:
  - Use `have`-expressions to prove the index is valid
  - Use `a[i]!` notation instead, runtime check is performed, and 'Panic' error
message is produced if index is not valid
  - Use `a[i]?` notation instead, result is an `Option` type
  - Use `a[i]'h` notation instead, where `h` is a proof that index is valid
⊢ idahoSpiders.inBounds 9
```

Because both the collection type and the index type are input parameters to the `GetElem` type class, new types can be used to index into existing collections. The positive number type `Pos` is a perfectly reasonable index into a `List`, with the caveat that it cannot point at the first entry. The follow instance of `GetElem` allows `Pos` to be used just as conveniently as `Nat` to find a list entry:

```
instance : GetElem (List α) Pos α (fun list n => list.length > n.toNat) where
  getElem (xs : List α) (i : Pos) ok := xs[i.toNat]
```

Indexing can also make sense for non-numeric indices. For example, `Bool` can be used to select between the fields in a point, with `false` corresponding to `x` and `true` corresponding to `y`:

```
instance : GetElem (PPoint α) Bool α (fun _ _ => True) where
  getElem (p : PPoint α) (i : Bool) _ :=
    if not i then p.x else p.y
```

In this case, both Booleans are valid indices. Because every possible `Bool` is in bounds, the evidence is simply the true proposition `True`.

Standard Classes

This section presents a variety of operators and functions that can be overloaded using type classes in Lean. Each operator or function corresponds to a method of a type class. Unlike C++, infix operators in Lean are defined as abbreviations for named functions; this means that overloading them for new types is not done using the operator itself, but rather using the underlying name (such as `HAdd.hAdd`).

Arithmetic

Most arithmetic operators are available in a heterogeneous form, where the arguments may have different type and an output parameter decides the type of the resulting expression. For each heterogeneous operator, there is a corresponding homogeneous version that can found by removing the letter `h`, so that `HAdd.hAdd` becomes `Add.add`. The following arithmetic operators are overloaded:

Expression	Desugaring	Class Name
`x + y`	`HAdd.hAdd x y`	`HAdd`
`x - y`	`HSub.hSub x y`	`HSub`
`x * y`	`HMul.hMul x y`	`HMul`
`x / y`	`HDiv.hDiv x y`	`HDiv`
`x % y`	`HMod.hMod x y`	`HMod`
`x ^ y`	`HPow.hPow x y`	`HPow`
`(- x)`	`Neg.neg x`	`Neg`

Bitwise Operators

Lean contains a number of standard bitwise operators that are overloaded using type classes. There are instances for fixed-width types such as `UInt8`, `UInt16`, `UInt32`, `UInt64`, and `USize`. The latter is the size of words on the current platform, typically 32 or 64 bits. The following bitwise operators are overloaded:

Expression	Desugaring	Class Name
`x &&& y`	`HAnd.hAnd x y`	`HAnd`

Expression	Desugaring	Class Name
`x ||| y`	`HOr.hOr x y`	`HOr`
`x ^^^ y`	`HXor.hXor x y`	`HXor`
`~~~ x`	`Complement.complement x`	`Complement`
`x >>> y`	`HShiftRight.hShiftRight x y`	`HShiftRight`
`x <<< y`	`HShiftLeft.hShiftLeft x y`	`HShiftLeft`

Because the names `And` and `Or` are already taken as the names of logical connectives, the homogeneous versions of `HAnd` and `HOr` are called `AndOp` and `OrOp` rather than `And` and `Or` .

Equality and Ordering

Testing equality of two values typically uses the `BEq` class, which is short for "Boolean equality". Due to Lean's use as a theorem prover, there are really two kinds of equality operators in Lean:

- *Boolean equality* is the same kind of equality that is found in other programming languages. It is a function that takes two values and returns a `Bool` . Boolean equality is written with two equals signs, just as in Python and C#. Because Lean is a pure functional language, there's no separate notions of reference vs value equality—pointers cannot be observed directly.
- *Propositional equality* is the mathematical statement that two things are equal. Propositional equality is not a function; rather, it is a mathematical statement that admits proof. It is written with a single equals sign. A statement of propositional equality is like a type that classifies evidence of this equality.

Both notions of equality are important, and used for different purposes. Boolean equality is useful in programs, when a decision needs to be made about whether two values are equal. For example, `"Octopus" ==  "Cuttlefish"` evaluates to `false` , and `"Octopodes" == "Octo".append "podes"` evaluates to `true` . Some values, such as functions, cannot be checked for equality. For example, `(fun (x : Nat) => 1 + x) == (Nat.succ ·)` yields the error:

```
failed to synthesize
  BEq (Nat → Nat)

Additional diagnostic information may be available using the `set_option
diagnostics true` command.
```

As this message indicates, `==` is overloaded using a type class. The expression `x == y` is actually shorthand for `BEq.beq x y`.

Propositional equality is a mathematical statement rather than an invocation of a program. Because propositions are like types that describe evidence for some statement, propositional equality has more in common with types like `String` and `Nat → List Int` than it does with Boolean equality. This means that it can't automatically be checked. However, the equality of any two expressions can be stated in Lean, so long as they have the same type. The statement `(fun (x : Nat) => 1 + x) = (Nat.succ ·)` is a perfectly reasonable statement. From the perspective of mathematics, two functions are equal if they map equal inputs to equal outputs, so this statement is even true, though it requires a one-line proof to convince Lean of this fact.

Generally speaking, when using Lean as a programming language, it's easiest to stick to Boolean functions rather than propositions. However, as the names `true` and `false` for `Bool`'s constructors suggest, this difference is sometimes blurred. Some propositions are *decidable*, which means that they can be checked just like a Boolean function. The function that checks whether the proposition is true or false is called a *decision procedure*, and it returns *evidence* of the truth or falsity of the proposition. Some examples of decidable propositions include equality and inequality of natural numbers, equality of strings, and "ands" and "ors" of propositions that are themselves decidable.

In Lean, `if` works with decidable propositions. For example, `2 < 4` is a proposition:

```
#check 2 < 4
```

```
2 < 4 : Prop
```

Nonetheless, it is perfectly acceptable to write it as the condition in an `if`. For example, `if 2 < 4 then 1 else 2` has type `Nat` and evaluates to `1`.

Not all propositions are decidable. If they were, then computers would be able to prove any true proposition just by running the decision procedure, and mathematicians would be out of a job. More specifically, decidable propositions have an instance of the `Decidable` type class, which contains the decision procedure. Trying to use a proposition that isn't decidable as if it were a `Bool` results in a failure to find the `Decidable` instance. For example, `if (fun (x : Nat) => 1 + x) = (Nat.succ ·) then "yes" else "no"` results in:

```
failed to synthesize
  Decidable ((fun x => 1 + x) = fun x => x.succ)

Additional diagnostic information may be available using the `set_option
diagnostics true` command.
```

The following propositions, that are usually decidable, are overloaded with type classes:

Expression	Desugaring	Class Name
`x < y`	`LT.lt x y`	`LT`
`x ≤ y`	`LE.le x y`	`LE`
`x > y`	`LT.lt y x`	`LT`
`x ≥ y`	`LE.le y x`	`LE`

Because defining new propositions hasn't yet been demonstrated, it may be difficult to define completely new instances of `LT` and `LE`. However, they can be defined in terms of existing instances. `LT` and `LE` instances for `Pos` can use the existing instances for `Nat`:

```
instance : LT Pos where
  lt x y := LT.lt x.toNat y.toNat

instance : LE Pos where
  le x y := LE.le x.toNat y.toNat
```

These propositions are not decidable by default because Lean doesn't unfold the definitions of propositions while synthesizing an instance. This can be bridged using the `inferInstanceAs` operator, which finds an instance for a given class if it exists:

```
instance {x : Pos} {y : Pos} : Decidable (x < y) :=
  inferInstanceAs (Decidable (x.toNat < y.toNat))

instance {x : Pos} {y : Pos} : Decidable (x ≤ y) :=
  inferInstanceAs (Decidable (x.toNat ≤ y.toNat))
```

The type checker confirms that the definitions of the propositions match. Confusing them results in an error:

```
instance {x : Pos} {y : Pos} : Decidable (x ≤ y) :=
  inferInstanceAs (Decidable (x.toNat < y.toNat))
```

```
type mismatch
  inferInstanceAs (Decidable (x.toNat < y.toNat))
has type
  Decidable (x.toNat < y.toNat) : Type
but is expected to have type
  Decidable (x ≤ y) : Type
```

Comparing values using `<`, `==`, and `>` can be inefficient. Checking first whether one value is less than another, and then whether they are equal, can require two traversals over large data structures. To solve this problem, Java and C# have standard `compareTo` and

`CompareTo` methods (respectively) that can be overridden by a class in order to implement all three operations at the same time. These methods return a negative integer if the receiver is less than the argument, zero if they are equal, and a positive integer if the receiver is greater than the argument. Rather than overloading the meaning of integers, Lean has a built-in inductive type that describes these three possibilities:

```
inductive Ordering where
  | lt
  | eq
  | gt
```

The `Ord` type class can be overloaded to produce these comparisons. For `Pos`, an implementation can be:

```
def Pos.comp : Pos → Pos → Ordering
  | Pos.one, Pos.one => Ordering.eq
  | Pos.one, Pos.succ _ => Ordering.lt
  | Pos.succ _, Pos.one => Ordering.gt
  | Pos.succ n, Pos.succ k => comp n k

instance : Ord Pos where
  compare := Pos.comp
```

In situations where `compareTo` would be the right approach in Java, use `Ord.compare` in Lean.

Hashing

Java and C# have `hashCode` and `GetHashCode` methods, respectively, that compute a hash of a value for use in data structures such as hash tables. The Lean equivalent is a type class called `Hashable`:

```
class Hashable (α : Type) where
  hash : α → UInt64
```

If two values are considered equal according to a `BEq` instance for their type, then they should have the same hashes. In other words, if `x == y` then `hash x == hash y`. If `x ≠ y`, then `hash x` won't necessarily differ from `hash y` (after all, there are infinitely more `Nat` values than there are `UInt64` values), but data structures built on hashing will have better performance if unequal values are likely to have unequal hashes. This is the same expectation as in Java and C#.

The standard library contains a function `mixHash` with type `UInt64 → UInt64 → UInt64` that can be used to combine hashes for different fields for a constructor. A reasonable hash function for an inductive datatype can be written by assigning a unique number to each constructor, and then mixing that number with the hashes of each field. For example, a `Hashable` instance for `Pos` can be written:

```
def hashPos : Pos → UInt64
  | Pos.one => 0
  | Pos.succ n => mixHash 1 (hashPos n)

instance : Hashable Pos where
  hash := hashPos
```

`Hashable` instances for polymorphic types can use recursive instance search. Hashing a `NonEmptyList α` is only possible when `α` can be hashed:

```
instance [Hashable α] : Hashable (NonEmptyList α) where
  hash xs := mixHash (hash xs.head) (hash xs.tail)
```

Binary trees use both recursion and recursive instance search in the implementations of `BEq` and `Hashable`:

```
inductive BinTree (α : Type) where
  | leaf : BinTree α
  | branch : BinTree α → α → BinTree α → BinTree α

def eqBinTree [BEq α] : BinTree α → BinTree α → Bool
  | BinTree.leaf, BinTree.leaf =>
    true
  | BinTree.branch l x r, BinTree.branch l2 x2 r2 =>
    x == x2 && eqBinTree l l2 && eqBinTree r r2
  | _, _ =>
    false

instance [BEq α] : BEq (BinTree α) where
  beq := eqBinTree

def hashBinTree [Hashable α] : BinTree α → UInt64
  | BinTree.leaf =>
    0
  | BinTree.branch left x right =>
    mixHash 1 (mixHash (hashBinTree left) (mixHash (hash x) (hashBinTree
right)))

instance [Hashable α] : Hashable (BinTree α) where
  hash := hashBinTree
```

Deriving Standard Classes

Instance of classes like `BEq` and `Hashable` are often quite tedious to implement by hand. Lean includes a feature called *instance deriving* that allows the compiler to automatically construct well-behaved instances of many type classes. In fact, the `deriving Repr` phrase in the definition of `Point` in the section on structures is an example of instance deriving.

Instances can be derived in two ways. The first can be used when defining a structure or inductive type. In this case, add `deriving` to the end of the type declaration followed by the names of the classes for which instances should be derived. For a type that is already defined, a standalone `deriving` command can be used. Write `deriving instance C1, C2, ... for T` to derive instances of `C1, C2, ...` for the type `T` after the fact.

`BEq` and `Hashable` instances can be derived for `Pos` and `NonEmptyList` using a very small amount of code:

```
deriving instance BEq, Hashable for Pos
deriving instance BEq, Hashable for NonEmptyList
```

Instances can be derived for at least the following classes:

- `Inhabited`
- `BEq`
- `Repr`
- `Hashable`
- `Ord`

In some cases, however, the derived `Ord` instance may not produce precisely the ordering desired in an application. When this is the case, it's fine to write an `Ord` instance by hand. The collection of classes for which instances can be derived can be extended by advanced users of Lean.

Aside from the clear advantages in programmer productivity and code readability, deriving instances also makes code easier to maintain, because the instances are updated as the definitions of types evolve. When reviewing changes to code, modifications that involve updates to datatypes are much easier to read without line after line of formulaic modifications to equality tests and hash computation.

Appending

Many datatypes have some sort of append operator. In Lean, appending two values is overloaded with the type class `HAppend`, which is a heterogeneous operation like that used for arithmetic operations:

```
class HAppend (α : Type) (β : Type) (γ : outParam Type) where
  hAppend : α → β → γ
```

The syntax `xs ++ ys` desugars to `HAppend.hAppend xs ys`. For homogeneous cases, it's enough to implement an instance of `Append`, which follows the usual pattern:

```
instance : Append (NonEmptyList α) where
  append xs ys :=
    { head := xs.head, tail := xs.tail ++ ys.head :: ys.tail }
```

After defining the above instance,

```
#eval idahoSpiders ++ idahoSpiders
```

has the following output:

```
{ head := "Banded Garden Spider",
tail := ["Long-legged Sac Spider",
         "Wolf Spider",
         "Hobo Spider",
         "Cat-faced Spider",
         "Banded Garden Spider",
         "Long-legged Sac Spider",
         "Wolf Spider",
         "Hobo Spider",
         "Cat-faced Spider"] }
```

Similarly, a definition of `HAppend` allows non-empty lists to be appended to ordinary lists:

```
instance : HAppend (NonEmptyList α) (List α) (NonEmptyList α) where
  hAppend xs ys :=
    { head := xs.head, tail := xs.tail ++ ys }
```

With this instance available,

```
#eval idahoSpiders ++ ["Trapdoor Spider"]
```

results in

```
{ head := "Banded Garden Spider",
  tail := ["Long-legged Sac Spider", "Wolf Spider", "Hobo Spider", "Cat-faced
Spider", "Trapdoor Spider"] }
```

Functors

A polymorphic type is a *functor* if it has an overload for a function named `map` that transforms every element contained in it by a function. While most languages use this terminology, C#'s equivalent of `map` is called `System.Linq.Enumerable.Select`. For example, mapping a function over a list constructs a new list in which each entry from the starting list has been replaced by the result of the function on that entry. Mapping a function `f` over an `Option` leaves `none` untouched, and replaces `some x` with `some (f x)`.

Here are some examples of functors and how their `Functor` instances overload `map`:

- `Functor.map (· + 5) [1, 2, 3]` evaluates to `[6, 7, 8]`
- `Functor.map toString (some (List.cons 5 List.nil))` evaluates to `some "[5]"`
- `Functor.map List.reverse [[1, 2, 3], [4, 5, 6]]` evaluates to `[[3, 2, 1], [6, 5, 4]]`

Because `Functor.map` is a bit of a long name for this common operation, Lean also provides an infix operator for mapping a function, namely `<$>`. The prior examples can be rewritten as follows:

- `(· + 5) <$> [1, 2, 3]` evaluates to `[6, 7, 8]`
- `toString <$> (some (List.cons 5 List.nil))` evaluates to `some "[5]"`
- `List.reverse <$> [[1, 2, 3], [4, 5, 6]]` evaluates to `[[3, 2, 1], [6, 5, 4]]`

An instance of `Functor` for `NonEmptyList` requires specifying the `map` function.

```
instance : Functor NonEmptyList where
  map f xs := { head := f xs.head, tail := f <$> xs.tail }
```

Here, `map` uses the `Functor` instance for `List` to map the function over the tail. This instance is defined for `NonEmptyList` rather than for `NonEmptyList α` because the argument type `α` plays no role in resolving the type class. A `NonEmptyList` can have a function mapped over it *no matter what the type of entries is*. If `α` were a parameter to the class, then it would be possible to make versions of `Functor` that only worked for `NonEmptyList Nat`, but part of being a functor is that `map` works for any entry type.

Here is an instance of `Functor` for `PPoint`:

```
instance : Functor PPoint where
  map f p := { x := f p.x, y := f p.y }
```

In this case, `f` has been applied to both `x` and `y`.

Even when the type contained in a functor is itself a functor, mapping a function only goes down one layer. That is, when using `map` on a `NonEmptyList (PPoint Nat)`, the function being mapped should take `PPoint Nat` as its argument rather than `Nat`.

The definition of the `Functor` class uses one more language feature that has not yet been discussed: default method definitions. Normally, a class will specify some minimal set of overloadable operations that make sense together, and then use polymorphic functions with instance implicit arguments that build on the overloaded operations to provide a larger library of features. For example, the function `concat` can concatenate any non-empty list whose entries are appendable:

```
def concat [Append α] (xs : NonEmptyList α) : α :=
  let rec catList (start : α) : List α → α
    | [] => start
    | (z :: zs) => catList (start ++ z) zs
  catList xs.head xs.tail
```

However, for some classes, there are operations that can be more efficiently implemented with knowledge of the internals of a datatype.

In these cases, a default method definition can be provided. A default method definition provides a default implementation of a method in terms of the other methods. However, instance implementors may choose to override this default with something more efficient. Default method definitions contain `:=` in a `class` definition.

In the case of `Functor`, some types have a more efficient way of implementing `map` when the function being mapped ignores its argument. Functions that ignore their arguments are called *constant functions* because they always return the same value. Here is the definition of `Functor`, in which `mapConst` has a default implementation:

```
class Functor (f : Type → Type) where
  map : {α β : Type} → (α → β) → f α → f β

  mapConst {α β : Type} (x : α) (coll : f β) : f α :=
    map (fun _ => x) coll
```

Just as a `Hashable` instance that doesn't respect `BEq` is buggy, a `Functor` instance that moves around the data as it maps the function is also buggy. For example, a buggy `Functor` instance for `List` might throw away its argument and always return the empty list, or it might reverse the list. A bad `Functor` instance for `PPoint` might place `f x` in both the `x` and the `y` fields, or swap them. Specifically, `Functor` instances should follow two rules:

1. Mapping the identity function should result in the original argument.
2. Mapping two composed functions should have the same effect as composing their

mapping.

More formally, the first rule says that `id <$> x` equals `x`. The second rule says that `map (fun y => f (g y)) x` equals `map f (map g x)`. The composition `fun y => f (g y)` can also be written `f ∘ g`. These rules prevent implementations of `map` that move the data around or delete some of it.

Messages You May Meet

Lean is not able to derive instances for all classes. For example, the code

```
deriving instance ToString for NonEmptyList
```

results in the following error:

```
default handlers have not been implemented yet, class: 'ToString' types:
[NonEmptyList]
```

Invoking `deriving instance` causes Lean to consult an internal table of code generators for type class instances. If the code generator is found, then it is invoked on the provided type to create the instance. This message, however, means that no code generator was found for `ToString`.

Exercises

- Write an instance of `HAppend (List α) (NonEmptyList α) (NonEmptyList α)` and test it.
- Implement a `Functor` instance for the binary tree datatype.

Coercions

In mathematics, it is common to use the same symbol to stand for different aspects of some object in different contexts. For example, if a ring is referred to in a context where a set is expected, then it is understood that the ring's underlying set is what's intended. In programming languages, it is common to have rules to automatically translate values of one type into values of another type. For instance, Java allows a `byte` to be automatically promoted to an `int`, and Kotlin allows a non-nullable type to be used in a context that expects a nullable version of the type.

In Lean, both purposes are served by a mechanism called *coercions*. When Lean encounters an expression of one type in a context that expects a different type, it will attempt to coerce the expression before reporting a type error. Unlike Java, C, and Kotlin, the coercions are extensible by defining instances of type classes.

Strings and Paths

In the source code to `feline`, a `String` is converted to a `FilePath` using the anonymous constructor syntax. In fact, this was not necessary: Lean defines a coercion from `String` to `FilePath`, so a string can be used in an position where a path is expected. Even though the function `IO.FS.readFile` has type `System.FilePath → IO String`, the following code is accepted by Lean:

```
def fileDumper : IO Unit := do
  let stdin ← IO.getStdin
  let stdout ← IO.getStdout
  stdout.putStr "Which file? "
  stdout.flush
  let f := (← stdin.getLine).trim
  stdout.putStrLn s!"'The file {f}' contains:"
  stdout.putStrLn (← IO.FS.readFile f)
```

`String.trim` removes leading and trailing whitespace from a string. On the last line of `fileDumper`, the coercion from `String` to `FilePath` automatically converts `f`, so it is not necessary to write `IO.FS.readFile ⟨f⟩`.

Positive Numbers

Every positive number corresponds to a natural number. The function `Pos.toNat` that was defined earlier converts a `Pos` to the corresponding `Nat`:

```
def Pos.toNat : Pos → Nat
  | Pos.one => 1
  | Pos.succ n => n.toNat + 1
```

The function `List.drop`, with type `{α : Type} → Nat → List α → List α`, removes a prefix of a list. Applying `List.drop` to a `Pos`, however, leads to a type error:

```
[1, 2, 3, 4].drop (2 : Pos)
```

```
application type mismatch
  List.drop 2
argument
  2
has type
  Pos : Type
but is expected to have type
  Nat : Type
```

Because the author of `List.drop` did not make it a method of a type class, it can't be overridden by defining a new instance.

The type class `Coe` describes overloaded ways of coercing from one type to another:

```
class Coe (α : Type) (β : Type) where
  coe : α → β
```

An instance of `Coe Pos Nat` is enough to allow the prior code to work:

```
instance : Coe Pos Nat where
  coe x := x.toNat

#eval [1, 2, 3, 4].drop (2 : Pos)
```

```
[3, 4]
```

Using `#check` shows the result of the instance search that was used behind the scenes:

```
#check [1, 2, 3, 4].drop (2 : Pos)
```

```
List.drop (Pos.toNat 2) [1, 2, 3, 4] : List Nat
```

Chaining Coercions

When searching for coercions, Lean will attempt to assemble a coercion out of a chain of smaller coercions. For example, there is already a coercion from `Nat` to `Int`. Because of that instance, combined with the `Coe Pos Nat` instance, the following code is accepted:

```
def oneInt : Int := Pos.one
```

This definition uses two coercions: from `Pos` to `Nat`, and then from `Nat` to `Int`.

The Lean compiler does not get stuck in the presence of circular coercions. For example, even if two types `A` and `B` can be coerced to one another, their mutual coercions can be used to find a path:

```
inductive A where
  | a

inductive B where
  | b

instance : Coe A B where
  coe _ := B.b

instance : Coe B A where
  coe _ := A.a

instance : Coe Unit A where
  coe _ := A.a

def coercedToB : B := ()
```

Remember: the double parentheses `()` is short for the constructor `Unit.unit`. After deriving a `Repr B` instance with `deriving instance Repr for B`,

```
#eval coercedToB
```

results in:

```
B.b
```

The `Option` type can be used similarly to nullable types in C# and Kotlin: the `none` constructor represents the absence of a value. The Lean standard library defines a coercion from any type `α` to `Option α` that wraps the value in `some`. This allows option types to be used in a manner even more similar to nullable types, because `some` can be omitted. For instance, the function `List.getLast?` that finds the last entry in a list can be written without a `some` around the return value `x`:

```
def List.last? : List α → Option α
  | [] => none
  | [x] => x
  | _ :: x :: xs => last? (x :: xs)
```

Instance search finds the coercion, and inserts a call to `coe`, which wraps the argument in `some`. These coercions can be chained, so that nested uses of `Option` don't require nested `some` constructors:

```
def perhapsPerhapsPerhaps : Option (Option (Option String)) :=
  "Please don't tell me"
```

Coercions are only activated automatically when Lean encounters a mismatch between an inferred type and a type that is imposed from the rest of the program. In cases with other errors, coercions are not activated. For example, if the error is that an instance is missing, coercions will not be used:

```
def perhapsPerhapsPerhapsNat : Option (Option (Option Nat)) :=
  392
```

```
failed to synthesize
  OfNat (Option (Option (Option Nat))) 392
numerals are polymorphic in Lean, but the numeral `392` cannot be used in a
context where the expected type is
  Option (Option (Option Nat))
due to the absence of the instance above

Additional diagnostic information may be available using the `set_option
diagnostics true` command.
```

This can be worked around by manually indicating the desired type to be used for `OfNat`:

```
def perhapsPerhapsPerhapsNat : Option (Option (Option Nat)) :=
  (392 : Nat)
```

Additionally, coercions can be manually inserted using an up arrow:

```
def perhapsPerhapsPerhapsNat : Option (Option (Option Nat)) :=
  ↑(392 : Nat)
```

In some cases, this can be used to ensure that Lean finds the right instances. It can also make the programmer's intentions more clear.

Non-Empty Lists and Dependent Coercions

An instance of `Coe α β` makes sense when the type `β` has a value that can represent each value from the type `α`. Coercing from `Nat` to `Int` makes sense, because the type `Int` contains all the natural numbers, but a coercion from `Int` to `Nat` is a poor idea because `Nat` does not contain the negative numbers. Similarly, a coercion from non-empty lists to ordinary lists makes sense because the `List` type can represent every non-empty list:

```
instance : Coe (NonEmptyList α) (List α) where
  coe
    | { head := x, tail := xs } => x :: xs
```

This allows non-empty lists to be used with the entire `List` API.

On the other hand, it is impossible to write an instance of `Coe (List α) (NonEmptyList α)`, because there's no non-empty list that can represent the empty list. This limitation can be worked around by using another version of coercions, which are called *dependent coercions*. Dependent coercions can be used when the ability to coerce from one type to another depends on which particular value is being coerced. Just as the `OfNat` type class takes the particular `Nat` being overloaded as a parameter, dependent coercion takes the value being coerced as a parameter:

```
class CoeDep (α : Type) (x : α) (β : Type) where
  coe : β
```

This is a chance to select only certain values, either by imposing further type class constraints on the value or by writing certain constructors directly. For example, any `List` that is not actually empty can be coerced to a `NonEmptyList`:

```
instance : CoeDep (List α) (x :: xs) (NonEmptyList α) where
  coe := { head := x, tail := xs }
```

Coercing to Types

In mathematics, it is common to have a concept that consists of a set equipped with additional structure. For example, a monoid is some set *S*, an element *s* of *S*, and an associative binary operator on *S*, such that *s* is neutral on the left and right of the operator. *S* is referred to as the "carrier set" of the monoid. The natural numbers with zero and addition form a monoid, because addition is associative and adding zero to any number is the identity. Similarly, the natural numbers with one and multiplication also form a monoid.

Monoids are also widely used in functional programming: lists, the empty list, and the append operator form a monoid, as do strings, the empty string, and string append:

```
structure Monoid where
  Carrier : Type
  neutral : Carrier
  op : Carrier → Carrier → Carrier

def natMulMonoid : Monoid :=
  { Carrier := Nat, neutral := 1, op := (· * ·) }

def natAddMonoid : Monoid :=
  { Carrier := Nat, neutral := 0, op := (· + ·) }

def stringMonoid : Monoid :=
  { Carrier := String, neutral := "", op := String.append }

def listMonoid (α : Type) : Monoid :=
  { Carrier := List α, neutral := [], op := List.append }
```

Given a monoid, it is possible to write the `foldMap` function that, in a single pass, transforms the entries in a list into a monoid's carrier set and then combines them using the monoid's operator. Because monoids have a neutral element, there is a natural result to return when the list is empty, and because the operator is associative, clients of the function don't have to care whether the recursive function combines elements from left to right or from right to left.

```
def foldMap (M : Monoid) (f : α → M.Carrier) (xs : List α) : M.Carrier :=
  let rec go (soFar : M.Carrier) : List α → M.Carrier
    | [] => soFar
    | y :: ys => go (M.op soFar (f y)) ys
  go M.neutral xs
```

Even though a monoid consists of three separate pieces of information, it is common to just refer to the monoid's name in order to refer to its set. Instead of saying "Let A be a monoid and let *x* and *y* be elements of its carrier set", it is common to say "Let *A* be a monoid and let *x* and *y* be elements of *A*". This practice can be encoded in Lean by defining a new kind of coercion, from the monoid to its carrier set.

The `CoeSort` class is just like the `Coe` class, with the exception that the target of the coercion must be a *sort*, namely `Type` or `Prop`. The term *sort* in Lean refers to these types that classify other types— `Type` classifies types that themselves classify data, and `Prop` classifies propositions that themselves classify evidence of their truth. Just as `Coe` is checked when a type mismatch occurs, `CoeSort` is used when something other than a sort is provided in a context where a sort would be expected.

The coercion from a monoid into its carrier set extracts the carrier:

```
instance : CoeSort Monoid Type where
  coe m := m.Carrier
```

With this coercion, the type signatures become less bureaucratic:

```
def foldMap (M : Monoid) (f : α → M) (xs : List α) : M :=
  let rec go (soFar : M) : List α → M
    | [] => soFar
    | y :: ys => go (M.op soFar (f y)) ys
  go M.neutral xs
```

Another useful example of `CoeSort` is used to bridge the gap between `Bool` and `Prop`. As discussed in the section on ordering and equality, Lean's `if` expression expects the condition to be a decidable proposition rather than a `Bool`. Programs typically need to be able to branch based on Boolean values, however. Rather than have two kinds of `if` expression, the Lean standard library defines a coercion from `Bool` to the proposition that the `Bool` in question is equal to `true`:

```
instance : CoeSort Bool Prop where
  coe b := b = true
```

In this case, the sort in question is `Prop` rather than `Type`.

Coercing to Functions

Many datatypes that occur regularly in programming consist of a function along with some extra information about it. For example, a function might be accompanied by a name to show in logs or by some configuration data. Additionally, putting a type in a field of a structure, similarly to the `Monoid` example, can make sense in contexts where there is more than one way to implement an operation and more manual control is needed than type classes would allow. For example, the specific details of values emitted by a JSON serializer may be important because another application expects a particular format. Sometimes, the function itself may be derivable from just the configuration data.

A type class called `CoeFun` can transform values from non-function types to function types. `CoeFun` has two parameters: the first is the type whose values should be transformed into functions, and the second is an output parameter that determines exactly which function type is being targeted.

```
class CoeFun (α : Type) (makeFunctionType : outParam (α → Type)) where
  coe : (x : α) → makeFunctionType x
```

The second parameter is itself a function that computes a type. In Lean, types are first-class and can be passed to functions or returned from them, just like anything else.

For example, a function that adds a constant amount to its argument can be represented as a wrapper around the amount to add, rather than by defining an actual function:

```
structure Adder where
  howMuch : Nat
```

A function that adds five to its argument has a `5` in the `howMuch` field:

```
def add5 : Adder := ⟨5⟩
```

This `Adder` type is not a function, and applying it to an argument results in an error:

```
#eval add5 3
```

```
function expected at
  add5
term has type
  Adder
```

Defining a `CoeFun` instance causes Lean to transform the adder into a function with type `Nat → Nat`:

```
instance : CoeFun Adder (fun _ => Nat → Nat) where
  coe a := (· + a.howMuch)

#eval add5 3
```

```
8
```

Because all `Adder`s should be transformed into `Nat → Nat` functions, the argument to `CoeFun`'s second parameter was ignored.

When the value itself is needed to determine the right function type, then `CoeFun`'s second parameter is no longer ignored. For example, given the following representation of JSON values:

```
inductive JSON where
  | true : JSON
  | false : JSON
  | null : JSON
  | string : String → JSON
  | number : Float → JSON
  | object : List (String × JSON) → JSON
  | array : List JSON → JSON
deriving Repr
```

a JSON serializer is a structure that tracks the type it knows how to serialize along with the serialization code itself:

```
structure Serializer where
  Contents : Type
  serialize : Contents → JSON
```

A serializer for strings need only wrap the provided string in the `JSON.string` constructor:

```
def Str : Serializer :=
  { Contents := String,
    serialize := JSON.string
  }
```

Viewing JSON serializers as functions that serialize their argument requires extracting the inner type of serializable data:

```
instance : CoeFun Serializer (fun s => s.Contents → JSON) where
  coe s := s.serialize
```

Given this instance, a serializer can be applied directly to an argument:

```
def buildResponse (title : String) (R : Serializer) (record : R.Contents) : JSON
:=
  JSON.object [
    ("title", JSON.string title),
    ("status", JSON.number 200),
    ("record", R record)
  ]
```

The serializer can be passed directly to `buildResponse`:

```
#eval buildResponse "Functional Programming in Lean" Str "Programming is fun!"
```

```
JSON.object
  [("title", JSON.string "Functional Programming in Lean"),
   ("status", JSON.number 200.000000),
   ("record", JSON.string "Programming is fun!")]
```

Aside: JSON as a String

It can be a bit difficult to understand JSON when encoded as Lean objects. To help make sure that the serialized response was what was expected, it can be convenient to write a simple converter from `JSON` to `String`. The first step is to simplify the display of numbers. `JSON` doesn't distinguish between integers and floating point numbers, and the type `Float` is used to represent both. In Lean, `Float.toString` includes a number of trailing zeros:

```
#eval (5 : Float).toString
```

```
"5.000000"
```

The solution is to write a little function that cleans up the presentation by dropping all trailing zeros, followed by a trailing decimal point:

```
def dropDecimals (numString : String) : String :=
  if numString.contains '.' then
    let noTrailingZeros := numString.dropRightWhile (· == '0')
    noTrailingZeros.dropRightWhile (· == '.')
  else numString
```

With this definition, `#eval dropDecimals (5 : Float).toString` yields `"5"`, and `#eval dropDecimals (5.2 : Float).toString` yields `"5.2"`.

The next step is to define a helper function to append a list of strings with a separator in between them:

```
def String.separate (sep : String) (strings : List String) : String :=
  match strings with
  | [] => ""
  | x :: xs => String.join (x :: xs.map (sep ++ ·))
```

This function is useful to account for comma-separated elements in JSON arrays and objects. `#eval ", ".separate ["1", "2"]` yields `"1, 2"`, `#eval ", ".separate ["1"]` yields `"1"`, and `#eval ", ".separate []` yields `""`. In the Lean standard library, this function is called `String.intercalate`.

Finally, a string escaping procedure is needed for JSON strings, so that the Lean string containing `"Hello!"` can be output as `"\"Hello!\""`. Fortunately, the Lean compiler contains an internal function for escaping JSON strings already, called `Lean.Json.escape`. To access this function, add `import Lean` to the beginning of your file.

The function that emits a string from a `JSON` value is declared `partial` because Lean cannot see that it terminates. This is because recursive calls to `asString` occur in functions that are being applied by `List.map`, and this pattern of recursion is complicated enough that Lean cannot see that the recursive calls are actually being performed on smaller values. In an application that just needs to produce JSON strings and doesn't need to mathematically reason about the process, having the function be `partial` is not likely to cause problems.

```
partial def JSON.asString (val : JSON) : String :=
  match val with
  | true => "true"
  | false => "false"
  | null => "null"
  | string s => "\"" ++ Lean.Json.escape s ++ "\""
  | number n => dropDecimals n.toString
  | object members =>
    let memberToString mem :=
      "\"" ++ Lean.Json.escape mem.fst ++ "\": " ++ asString mem.snd
    "{" ++ ", ".separate (members.map memberToString) ++ "}"
  | array elements =>
    "[" ++ ", ".separate (elements.map asString) ++ "]"
```

With this definition, the output of serialization is easier to read:

```
#eval (buildResponse "Functional Programming in Lean" Str "Programming is
fun!").asString
```

```
"{\\"title\\": \\"Functional Programming in Lean\\", \\"status\\": 200, \
\"record\\": \\"Programming is fun!\\"}"
```

Messages You May Meet

Natural number literals are overloaded with the `OfNat` type class. Because coercions fire in cases where types don't match, rather than in cases of missing instances, a missing `OfNat` instance for a type does not cause a coercion from `Nat` to be applied:

```
def perhapsPerhapsPerhapsNat : Option (Option (Option Nat)) :=
  392
```

```
failed to synthesize
  OfNat (Option (Option (Option Nat))) 392
numerals are polymorphic in Lean, but the numeral `392` cannot be used in a
context where the expected type is
  Option (Option (Option Nat))
due to the absence of the instance above

Additional diagnostic information may be available using the `set_option
diagnostics true` command.
```

Design Considerations

Coercions are a powerful tool that should be used responsibly. On the one hand, they can allow an API to naturally follow the everyday rules of the domain being modeled. This can be the difference between a bureaucratic mess of manual conversion functions and a clear program. As Abelson and Sussman wrote in the preface to *Structure and Interpretation of Computer Programs* (MIT Press, 1996),

> Programs must be written for people to read, and only incidentally for machines to execute.

Coercions, used wisely, are a valuable means of achieving readable code that can serve as the basis for communication with domain experts. APIs that rely heavily on coercions have a number of important limitations, however. Think carefully about these limitations before using coercions in your own libraries.

First off, coercions are only applied in contexts where enough type information is available for Lean to know all of the types involved, because there are no output parameters in the coercion type classes. This means that a return type annotation on a function can be the difference between a type error and a successfully applied coercion. For example, the coercion from non-empty lists to lists makes the following program work:

```
def lastSpider : Option String :=
  List.getLast? idahoSpiders
```

On the other hand, if the type annotation is omitted, then the result type is unknown, so Lean is unable to find the coercion:

```
def lastSpider :=
  List.getLast? idahoSpiders
```

```
application type mismatch
  List.getLast? idahoSpiders
argument
  idahoSpiders
has type
  NonEmptyList String : Type
but is expected to have type
  List ?m.25493 : Type
```

More generally, when a coercion is not applied for some reason, the user receives the original type error, which can make it difficult to debug chains of coercions.

Finally, coercions are not applied in the context of field accessor notation. This means that there is still an important difference between expressions that need to be coerced and those that don't, and this difference is visible to users of your API.

Additional Conveniences

Constructor Syntax for Instances

Behind the scenes, type classes are structure types and instances are values of these types. The only differences are that Lean stores additional information about type classes, such as which parameters are output parameters, and that instances are registered for searching. While values that have structure types are typically defined using either `⟨...⟩` syntax or with braces and fields, and instances are typically defined using `where`, both syntaxes work for both kinds of definition.

For example, a forestry application might represent trees as follows:

```
structure Tree : Type where
  latinName : String
  commonNames : List String

def oak : Tree :=
  ⟨"Quercus robur", ["common oak", "European oak"]⟩

def birch : Tree :=
  { latinName := "Betula pendula",
    commonNames := ["silver birch", "warty birch"]
  }

def sloe : Tree where
  latinName := "Prunus spinosa"
  commonNames := ["sloe", "blackthorn"]
```

All three syntaxes are equivalent.

Similarly, type class instances can be defined using all three syntaxes:

```
class Display (α : Type) where
  displayName : α → String

instance : Display Tree :=
  ⟨Tree.latinName⟩

instance : Display Tree :=
  { displayName := Tree.latinName }

instance : Display Tree where
  displayName t := t.latinName
```

The `where` syntax is typically used for instances, while structures use either the curly-brace syntax or the `where` syntax. The `⟨...⟩` syntax can be useful when emphasizing that a structure type is very much like a tuple in which the fields happen to be named, but the names are not important at the moment. However, there are situations where it can make sense to use other alternatives. In particular, a library might provide a function that constructs an instance value. Placing a call to this function after `:=` in an instance declaration is the easiest way to use such a function.

Examples

When experimenting with Lean code, definitions can be more convenient to use than `#eval` or `#check` commands. First off, definitions don't produce any output, which can help keep the reader's focus on the most interesting output. Secondly, it's easiest to write most Lean programs by starting with a type signature, allowing Lean to provide more assistance and better error messages while writing the program itself. On the other hand, `#eval` and `#check` are easiest to use in contexts where Lean is able to determine the type from the provided expression. Thirdly, `#eval` cannot be used with expressions whose types don't have `ToString` or `Repr` instances, such as functions. Finally, multi-step `do` blocks, `let`-expressions, and other syntactic forms that take multiple lines are particularly difficult to write with a type annotation in `#eval` or `#check`, simply because the required parenthesization can be difficult to predict.

To work around these issues, Lean supports the explicit indication of examples in a source file. An example is like a definition without a name. For instance, a non-empty list of birds commonly found in Copenhagen's green spaces can be written:

```
example : NonEmptyList String :=
  { head := "Sparrow",
    tail := ["Duck", "Swan", "Magpie", "Eurasian coot", "Crow"]
  }
```

Examples may define functions by accepting arguments:

```
example (n : Nat) (k : Nat) : Bool :=
  n + k == k + n
```

While this creates a function behind the scenes, this function has no name and cannot be called. Nonetheless, this is useful for demonstrating how a library can be used with arbitrary or unknown values of some given type. In source files, `example` declarations are best paired with comments that explain how the example illustrates the concepts of the library.

Summary

Type Classes and Overloading

Type classes are Lean's mechanism for overloading functions and operators. A polymorphic function can be used with multiple types, but it behaves in the same manner no matter which type it is used with. For example, a polymorphic function that appends two lists can be used no matter the type of the entries in the list, but it is unable to have different behavior depending on which particular type is found. An operation that is overloaded with type classes, on the other hand, can also be used with multiple types. However, each type requires its own implementation of the overloaded operation. This means that the behavior can vary based on which type is provided.

A *type class* has a name, parameters, and a body that consists of a number of names with types. The name is a way to refer to the overloaded operations, the parameters determine which aspects of the definitions can be overloaded, and the body provides the names and type signatures of the overloadable operations. Each overloadable operation is called a *method* of the type class. Type classes may provide default implementations of some methods in terms of the others, freeing implementors from defining each overload by hand when it is not needed.

An *instance* of a type class provides implementations of the methods for given parameters. Instances may be polymorphic, in which case they can work for a variety of parameters, and they may optionally provide more specific implementations of default methods in cases where a more efficient version exists for some particular type.

Type class parameters are either *input parameters* (the default), or *output parameters* (indicated by an `outParam` modifier). Lean will not begin searching for an instance until all input parameters are no longer metavariables, while output parameters may be solved while searching for instances. Parameters to a type class need not be types—they may also be ordinary values. The `OfNat` type class, used to overload natural number literals, takes the overloaded `Nat` itself as a parameter, which allows instances to restrict the allowed numbers.

Instances may be marked with a `@[default_instance]` attribute. When an instance is a default instance, then it will be chosen as a fallback when Lean would otherwise fail to find an instance due to the presence of metavariables in the type.

Type Classes for Common Syntax

Most infix operators in Lean are overridden with a type class. For instance, the addition operator corresponds to a type class called `Add`. Most of these operators have a corresponding heterogeneous version, in which the two arguments need not have the same type. These heterogenous operators are overloaded using a version of the class whose name starts with `H`, such as `HAdd`.

Indexing syntax is overloaded using a type class called `GetElem`, which involves proofs. `GetElem` has two output parameters, which are the type of elements to be extracted from the collection and a function that can be used to determine what counts as evidence that the index value is in bounds for the collection. This evidence is described by a proposition, and Lean attempts to prove this proposition when array indexing is used. When Lean is unable to check that list or array access operations are in bounds at compile time, the check can be deferred to run time by appending a `?` to the indexing operation.

Functors

A functor is a polymorphic type that supports a mapping operation. This mapping operation transforms all elements "in place", changing no other structure. For instance, lists are functors and the mapping operation may neither drop, duplicate, nor mix up entries in the list.

While functors are defined by having `map`, the `Functor` type class in Lean contains an additional default method that is responsible for mapping the constant function over a value, replacing all values whose type are given by polymorphic type variable with the same new value. For some functors, this can be done more efficiently than traversing the entire structure.

Deriving Instances

Many type classes have very standard implementations. For instance, the Boolean equality class `BEq` is usually implemented by first checking whether both arguments are built with the same constructor, and then checking whether all their arguments are equal. Instances for these classes can be created *automatically*.

When defining an inductive type or a structure, a `deriving` clause at the end of the declaration will cause instances to be created automatically. Additionally, the `deriving instance ... for ...` command can be used outside of the definition of a datatype to cause an instance to be generated. Because each class for which instances can be derived requires special handling, not all classes are derivable.

Coercions

Coercions allow Lean to recover from what would normally be a compile-time error by inserting a call to a function that transforms data from one type to another. For example, the coercion from any type `α` to the type `Option α` allows values to be written directly, rather than with the `some` constructor, making `Option` work more like nullable types from object-oriented languages.

There are multiple kinds of coercion. They can recover from different kinds of errors, and they are represented by their own type classes. The `Coe` class is used to recover from type errors. When Lean has an expression of type `α` in a context that expects something with type `β`, Lean first attempts to string together a chain of coercions that can transform `α`s into `β`s, and only displays the error when this cannot be done. The `CoeDep` class takes the specific value being coerced as an extra parameter, allowing either further type class search to be done on the value or allowing constructors to be used in the instance to limit the scope of the conversion. The `CoeFun` class intercepts what would otherwise be a "not a function" error when compiling a function application, and allows the value in the function position to be transformed into an actual function if possible.

Monads

In C# and Kotlin, the `?.` operator is a way to look up a property or call a method on a potentially-null value. If the receiver is `null`, the whole expression is null. Otherwise, the underlying non-`null` value receives the call. Uses of `?.` can be chained, in which case the first `null` result terminates the chain of lookups. Chaining null-checks like this is much more convenient than writing and maintaining deeply nested `if`s.

Similarly, exceptions are significantly more convenient than manually checking and propagating error codes. At the same time, logging is easiest to accomplish by having a dedicated logging framework, rather than having each function return both its log results and its return value. Chained null checks and exceptions typically require language designers to anticipate this use case, while logging frameworks typically make use of side effects to decouple code that logs from the accumulation of the logs.

All these features and more can be implemented in library code as instances of a common API called `Monad`. Lean provides dedicated syntax that makes this API convenient to use, but can also get in the way of understanding what is going on behind the scenes. This chapter begins with the nitty-gritty presentation of manually nesting null checks, and builds from there to the convenient, general API. Please suspend your disbelief in the meantime.

Checking for `none`: Don't Repeat Yourself

In Lean, pattern matching can be used to chain checks for null. Getting the first entry from a list can just use the optional indexing notation:

```
def first (xs : List α) : Option α :=
  xs[0]?
```

The result must be an `Option` because empty lists have no first entry. Extracting the first and third entries requires a check that each is not `none`:

```
def firstThird (xs : List α) : Option (α × α) :=
  match xs[0]? with
  | none => none
  | some first =>
    match xs[2]? with
    | none => none
    | some third =>
      some (first, third)
```

Similarly, extracting the first, third, and fifth entries requires more checks that the values are not `none`:

```
def firstThirdFifth (xs : List α) : Option (α × α × α) :=
  match xs[0]? with
  | none => none
  | some first =>
    match xs[2]? with
    | none => none
    | some third =>
      match xs[4]? with
      | none => none
      | some fifth =>
        some (first, third, fifth)
```

And adding the seventh entry to this sequence begins to become quite unmanageable:

```
def firstThirdFifthSeventh (xs : List α) : Option (α × α × α × α) :=
  match xs[0]? with
  | none => none
  | some first =>
    match xs[2]? with
    | none => none
    | some third =>
      match xs[4]? with
      | none => none
      | some fifth =>
        match xs[6]? with
        | none => none
        | some seventh =>
          some (first, third, fifth, seventh)
```

The fundamental problem with this code is that it addresses two concerns: extracting the numbers and checking that all of them are present, but the second concern is addressed by copying and pasting the code that handles the `none` case. It is often good style to lift a repetitive segment into a helper function:

```
def andThen (opt : Option α) (next : α → Option β) : Option β :=
  match opt with
  | none => none
  | some x => next x
```

This helper, which is used similarly to `?.` in C# and Kotlin, takes care of propagating `none` values. It takes two arguments: an optional value and a function to apply when the value is not `none`. If the first argument is `none`, then the helper returns `none`. If the first argument is not `none`, then the function is applied to the contents of the `some` constructor.

Now, `firstThird` can be rewritten to use `andThen` instead of pattern matching:

```
def firstThird (xs : List α) : Option (α × α) :=
  andThen xs[0]? fun first =>
  andThen xs[2]? fun third =>
  some (first, third)
```

In Lean, functions don't need to be enclosed in parentheses when passed as arguments. The following equivalent definition uses more parentheses and indents the bodies of functions:

```
def firstThird (xs : List α) : Option (α × α) :=
  andThen xs[0]? (fun first =>
    andThen xs[2]? (fun third =>
      some (first, third)))
```

The `andThen` helper provides a sort of "pipeline" through which values flow, and the version with the somewhat unusual indentation is more suggestive of this fact. Improving the syntax used to write `andThen` can make these computations even easier to understand.

Infix Operators

In Lean, infix operators can be declared using the `infix`, `infixl`, and `infixr` commands, which create (respectively) non-associative, left-associative, and right-associative operators. When used multiple times in a row, a *left associative* operator stacks up the opening parentheses on the left side of the expression. The addition operator `+` is left associative, so `w + x + y + z` is equivalent to `(((w + x) + y) + z)`. The exponentiation operator `^` is right associative, so `w ^ x ^ y ^ z` is equivalent to `(w ^ (x ^ (y ^ z)))`. Comparison operators such as `<` are non-associative, so `x < y < z` is a syntax error and requires manual parentheses.

The following declaration makes `andThen` into an infix operator:

```
infixl:55 " ~~> " => andThen
```

The number following the colon declares the *precedence* of the new infix operator. In ordinary mathematical notation, `x + y * z` is equivalent to `x + (y * z)` even though both `+` and `*` are left associative. In Lean, `+` has precedence 65 and `*` has precedence 70. Higher-precedence operators are applied before lower-precedence operators. According to the declaration of `~~>`, both `+` and `*` have higher precedence, and thus apply first. Typically, figuring out the most convenient precedences for a group of operators requires some experimentation and a large collection of examples.

Following the new infix operator is a double arrow `=>`, which specifies the named function to be used for the infix operator. Lean's standard library uses this feature to define `+` and `*` as infix operators that point at `HAdd.hAdd` and `HMul.hMul`, respectively, allowing type classes to be used to overload the infix operators. Here, however, `andThen` is just an ordinary function.

Having defined an infix operator for `andThen`, `firstThird` can be rewritten in a way that brings the "pipeline" feeling of `none`-checks front and center:

```
def firstThirdInfix (xs : List α) : Option (α × α) :=
  xs[0]? ~~> fun first =>
  xs[2]? ~~> fun third =>
  some (first, third)
```

This style is much more concise when writing larger functions:

```
def firstThirdFifthSeventh (xs : List α) : Option (α × α × α × α) :=
  xs[0]? ~~> fun first =>
  xs[2]? ~~> fun third =>
  xs[4]? ~~> fun fifth =>
  xs[6]? ~~> fun seventh =>
  some (first, third, fifth, seventh)
```

Propagating Error Messages

Pure functional languages such as Lean have no built-in exception mechanism for error handling, because throwing or catching an exception is outside of the step-by-step evaluation model for expressions. However, functional programs certainly need to handle errors. In the case of `firstThirdFifthSeventh`, it is likely relevant for a user to know just how long the list was and where the lookup failed.

This is typically accomplished by defining a datatype that can be either an error or a result, and translating functions with exceptions into functions that return this datatype:

```
inductive Except (ε : Type) (α : Type) where
  | error : ε → Except ε α
  | ok : α → Except ε α
deriving BEq, Hashable, Repr
```

The type variable `ε` stands for the type of errors that can be produced by the function. Callers are expected to handle both errors and successes, which makes the type variable `ε` play a role that is a bit like that of a list of checked exceptions in Java.

Similarly to `Option`, `Except` can be used to indicate a failure to find an entry in a list. In this case, the error type is a `String`:

```
def get (xs : List α) (i : Nat) : Except String α :=
  match xs[i]? with
  | none => Except.error s!"Index {i} not found (maximum is {xs.length - 1})"
  | some x => Except.ok x
```

Looking up an in-bounds value yields an `Except.ok`:

```
def ediblePlants : List String :=
  ["ramsons", "sea plantain", "sea buckthorn", "garden nasturtium"]

#eval get ediblePlants 2
```

```
Except.ok "sea buckthorn"
```

Looking up an out-of-bounds value yields an `Except.error`:

```
#eval get ediblePlants 4
```

```
Except.error "Index 4 not found (maximum is 3)"
```

A single list lookup can conveniently return a value or an error:

```
def first (xs : List α) : Except String α :=
  get xs 0
```

However, performing two list lookups requires handling potential failures:

```
def firstThird (xs : List α) : Except String (α × α) :=
  match get xs 0 with
  | Except.error msg => Except.error msg
  | Except.ok first =>
    match get xs 2 with
    | Except.error msg => Except.error msg
    | Except.ok third =>
      Except.ok (first, third)
```

Adding another list lookup to the function requires still more error handling:

```
def firstThirdFifth (xs : List α) : Except String (α × α × α) :=
  match get xs 0 with
  | Except.error msg => Except.error msg
  | Except.ok first =>
    match get xs 2 with
    | Except.error msg => Except.error msg
    | Except.ok third =>
      match get xs 4 with
      | Except.error msg => Except.error msg
      | Except.ok fifth =>
        Except.ok (first, third, fifth)
```

And one more list lookup begins to become quite unmanageable:

```
def firstThirdFifthSeventh (xs : List α) : Except String (α × α × α × α) :=
  match get xs 0 with
  | Except.error msg => Except.error msg
  | Except.ok first =>
    match get xs 2 with
    | Except.error msg => Except.error msg
    | Except.ok third =>
      match get xs 4 with
      | Except.error msg => Except.error msg
      | Except.ok fifth =>
        match get xs 6 with
        | Except.error msg => Except.error msg
        | Except.ok seventh =>
          Except.ok (first, third, fifth, seventh)
```

Once again, a common pattern can be factored out into a helper. Each step through the function checks for an error, and only proceeds with the rest of the computation if the result was a success. A new version of `andThen` can be defined for `Except`:

```
def andThen (attempt : Except e α) (next : α → Except e β) : Except e β :=
  match attempt with
  | Except.error msg => Except.error msg
  | Except.ok x => next x
```

Just as with `Option`, this version of `andThen` allows a more concise definition of `firstThird`:

```
def firstThird' (xs : List α) : Except String (α × α) :=
  andThen (get xs 0) fun first  =>
  andThen (get xs 2) fun third =>
  Except.ok (first, third)
```

In both the `Option` and `Except` case, there are two repeating patterns: there is the checking of intermediate results at each step, which has been factored out into `andThen`, and there is the final successful result, which is `some` or `Except.ok`, respectively. For the sake of convenience, success can be factored out into a helper called `ok`:

```
def ok (x : α) : Except ε α := Except.ok x
```

Similarly, failure can be factored out into a helper called `fail`:

```
def fail (err : ε) : Except ε α := Except.error err
```

Using `ok` and `fail` makes `get` a little more readable:

```
def get (xs : List α) (i : Nat) : Except String α :=
  match xs[i]? with
  | none => fail s!"Index {i} not found (maximum is {xs.length - 1})"
  | some x => ok x
```

After adding the infix declaration for `andThen`, `firstThird` can be just as concise as the version that returns an `Option`:

```
infixl:55 " ~~> " => andThen

def firstThird (xs : List α) : Except String (α × α) :=
  get xs 0 ~~> fun first =>
  get xs 2 ~~> fun third =>
  ok (first, third)
```

The technique scales similarly to larger functions:

```
def firstThirdFifthSeventh (xs : List α) : Except String (α × α × α × α) :=
  get xs 0 ~~> fun first =>
  get xs 2 ~~> fun third =>
  get xs 4 ~~> fun fifth =>
  get xs 6 ~~> fun seventh =>
  ok (first, third, fifth, seventh)
```

Logging

A number is even if dividing it by 2 leaves no remainder:

```
def isEven (i : Int) : Bool :=
  i % 2 == 0
```

The function `sumAndFindEvens` computes the sum of a list while remembering the even numbers encountered along the way:

```
def sumAndFindEvens : List Int → List Int × Int
  | [] => ([], 0)
  | i :: is =>
    let (moreEven, sum) := sumAndFindEvens is
    (if isEven i then i :: moreEven else moreEven, sum + i)
```

This function is a simplified example of a common pattern. Many programs need to traverse a data structure once, while both computing a main result and accumulating some kind of tertiary extra result. One example of this is logging: a program that is an `IO` action can always log to a file on disk, but because the disk is outside of the mathematical world of Lean functions, it becomes much more difficult to prove things about logs based on `IO`. Another example is a function that computes the sum of all the nodes in a tree with an inorder traversal, while simultaneously recording each nodes visited:

```
def inorderSum : BinTree Int → List Int × Int
  | BinTree.leaf => ([], 0)
  | BinTree.branch l x r =>
    let (leftVisited, leftSum) := inorderSum l
    let (hereVisited, hereSum) := ([x], x)
    let (rightVisited, rightSum) := inorderSum r
    (leftVisited ++ hereVisited ++ rightVisited, leftSum + hereSum + rightSum)
```

Both `sumAndFindEvens` and `inorderSum` have a common repetitive structure. Each step of computation returns a pair that consists of a list of data that have been saved along with the primary result. The lists are then appended, and the primary result is computed and paired with the appended lists. The common structure becomes more apparent with a small rewrite of `sumAndFindEvens` that more cleanly separates the concerns of saving even numbers and computing the sum:

```
def sumAndFindEvens : List Int → List Int × Int
  | [] => ([], 0)
  | i :: is =>
    let (moreEven, sum) := sumAndFindEvens is
    let (evenHere, ()) := (if isEven i then [i] else [], ())
    (evenHere ++ moreEven, sum + i)
```

For the sake of clarity, a pair that consists of an accumulated result together with a value can be given its own name:

```
structure WithLog (logged : Type) (α : Type) where
  log : List logged
  val : α
```

Similarly, the process of saving a list of accumulated results while passing a value on to the next step of a computation can be factored out into a helper, once again named `andThen`:

```
def andThen (result : WithLog α β) (next : β → WithLog α γ) : WithLog α γ :=
  let {log := thisOut, val := thisRes} := result
  let {log := nextOut, val := nextRes} := next thisRes
  {log := thisOut ++ nextOut, val := nextRes}
```

In the case of errors, `ok` represents an operation that always succeeds. Here, however, it is an operation that simply returns a value without logging anything:

```
def ok (x : β) : WithLog α β := {log := [], val := x}
```

Just as `Except` provides `fail` as a possibility, `WithLog` should allow items to be added to a log. This has no interesting return value associated with it, so it returns `Unit`:

```
def save (data : α) : WithLog α Unit :=
  {log := [data], val := ()}
```

`WithLog`, `andThen`, `ok`, and `save` can be used to separate the logging concern from the summing concern in both programs:

```
def sumAndFindEvens : List Int → WithLog Int Int
  | [] => ok 0
  | i :: is =>
    andThen (if isEven i then save i else ok ()) fun () =>
    andThen (sumAndFindEvens is) fun sum =>
    ok (i + sum)

def inorderSum : BinTree Int → WithLog Int Int
  | BinTree.leaf => ok 0
  | BinTree.branch l x r =>
    andThen (inorderSum l) fun leftSum =>
    andThen (save x) fun () =>
    andThen (inorderSum r) fun rightSum =>
    ok (leftSum + x + rightSum)
```

And, once again, the infix operator helps put focus on the correct steps:

```
infixl:55 " ~~> " => andThen

def sumAndFindEvens : List Int → WithLog Int Int
  | [] => ok 0
  | i :: is =>
    (if isEven i then save i else ok ()) ~~> fun () =>
    sumAndFindEvens is ~~> fun sum =>
    ok (i + sum)

def inorderSum : BinTree Int → WithLog Int Int
  | BinTree.leaf => ok 0
  | BinTree.branch l x r =>
    inorderSum l ~~> fun leftSum =>
    save x ~~> fun () =>
    inorderSum r ~~> fun rightSum =>
    ok (leftSum + x + rightSum)
```

Numbering Tree Nodes

An *inorder numbering* of a tree associates each data point in the tree with the step it would be visited at in an inorder traversal of the tree. For example, consider `aTree`:

```
open BinTree in
def aTree :=
  branch
    (branch
       (branch leaf "a" (branch leaf "b" leaf))
       "c"
       leaf)
    "d"
    (branch leaf "e" leaf)
```

Its inorder numbering is:

```
BinTree.branch
  (BinTree.branch
    (BinTree.branch (BinTree.leaf) (0, "a") (BinTree.branch (BinTree.leaf) (1,
"b") (BinTree.leaf)))
    (2, "c")
    (BinTree.leaf))
  (3, "d")
  (BinTree.branch (BinTree.leaf) (4, "e") (BinTree.leaf))
```

Trees are most naturally processed with recursive functions, but the usual pattern of recursion on trees makes it difficult to compute an inorder numbering. This is because the highest number assigned anywhere in the left subtree is used to determine the numbering of a node's data value, and then again to determine the starting point for numbering the right subtree. In an imperative language, this issue can be worked around by using a mutable variable that contains the next number to be assigned. The following Python program computes an inorder numbering using a mutable variable:

```
class Branch:
    def __init__(self, value, left=None, right=None):
        self.left = left
        self.value = value
        self.right = right
    def __repr__(self):
        return f'Branch({self.value!r}, left={self.left!r}, right={self.right!
r})'

def number(tree):
    num = 0
    def helper(t):
        nonlocal num
        if t is None:
            return None
        else:
            new_left = helper(t.left)
            new_value = (num, t.value)
            num += 1
            new_right = helper(t.right)
            return Branch(left=new_left, value=new_value, right=new_right)

    return helper(tree)
```

The numbering of the Python equivalent of `aTree` is:

```
a_tree = Branch("d",
                left=Branch("c",
                            left=Branch("a", left=None, right=Branch("b")),
                            right=None),
                right=Branch("e"))
```

and its numbering is:

```
>>> number(a_tree)
Branch((3, 'd'), left=Branch((2, 'c'), left=Branch((0, 'a'), left=None,
right=Branch((1, 'b'), left=None, right=None)), right=None), right=Branch((4,
'e'), left=None, right=None))
```

Even though Lean does not have mutable variables, a workaround exists. From the point of view of the rest of the world, the mutable variable can be thought of as having two relevant aspects: its value when the function is called, and its value when the function returns. In other words, a function that uses a mutable variable can be seen as a function that takes the mutable variable's starting value as an argument, returning a pair of the variable's final value and the function's result. This final value can then be passed as an argument to the next step.

Just as the Python example uses an outer function that establishes a mutable variable and an inner helper function that changes the variable, a Lean version of the function uses an outer function that provides the variable's starting value and explicitly returns the function's result along with an inner helper function that threads the variable's value while computing the numbered tree:

```
def number (t : BinTree α) : BinTree (Nat × α) :=
  let rec helper (n : Nat) : BinTree α → (Nat × BinTree (Nat × α))
    | BinTree.leaf => (n, BinTree.leaf)
    | BinTree.branch left x right =>
      let (k, numberedLeft) := helper n left
      let (i, numberedRight) := helper (k + 1) right
      (i, BinTree.branch numberedLeft (k, x) numberedRight)
  (helper 0 t).snd
```

This code, like the `none`-propagating `Option` code, the `error`-propagating `Except` code, and the log-accumulating `WithLog` code, commingles two concerns: propagating the value of the counter, and actually traversing the tree to find the result. Just as in those cases, an `andThen` helper can be defined to propagate state from one step of a computation to another. The first step is to give a name to the pattern of taking an input state as an argument and returning an output state together with a value:

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

In the case of `State`, `ok` is a function that returns the input state unchanged, along with the provided value:

```
def ok (x : α) : State σ α :=
  fun s => (s, x)
```

When working with a mutable variable, there are two fundamental operations: reading the value and replacing it with a new one. Reading the current value is accomplished with a function that places the input state unmodified into the output state, and also places it into the value field:

```
def get : State σ σ :=
  fun s => (s, s)
```

Writing a new value consists of ignoring the input state, and placing the provided new value into the output state:

```
def set (s : σ) : State σ Unit :=
  fun _ => (s, ())
```

Finally, two computations that use state can be sequenced by finding both the output state and return value of the first function, then passing them both into the next function:

```
def andThen (first : State σ α) (next : α → State σ β) : State σ β :=
  fun s =>
    let (s', x) := first s
    next x s'

infixl:55 " ~~> " => andThen
```

Using `State` and its helpers, local mutable state can be simulated:

```
def number (t : BinTree α) : BinTree (Nat × α) :=
  let rec helper : BinTree α → State Nat (BinTree (Nat × α))
    | BinTree.leaf => ok BinTree.leaf
    | BinTree.branch left x right =>
      helper left ~~> fun numberedLeft =>
      get ~~> fun n =>
      set (n + 1) ~~> fun () =>
      helper right ~~> fun numberedRight =>
      ok (BinTree.branch numberedLeft (n, x) numberedRight)
  (helper t 0).snd
```

Because `State` simulates only a single local variable, `get` and `set` don't need to refer to any particular variable name.

Monads: A Functional Design Pattern

Each of these examples has consisted of:

- A polymorphic type, such as `Option`, `Except ε`, `WithLog logged`, or `State σ`
- An operator `andThen` that takes care of some repetitive aspect of sequencing programs that have this type
- An operator `ok` that is (in some sense) the most boring way to use the type
- A collection of other operations, such as `none`, `fail`, `save`, and `get`, that name ways of using the type

This style of API is called a *monad*. While the idea of monads is derived from a branch of mathematics called category theory, no understanding of category theory is needed in order to use them for programming. The key idea of monads is that each monad encodes a particular kind of side effect using the tools provided by the pure functional language Lean. For example, `Option` represents programs that can fail by returning `none`, `Except` represents programs that can throw exceptions, `WithLog` represents programs that accumulate a log while running, and `State` represents programs with a single mutable variable.

The Monad Type Class

Rather than having to import an operator like `ok` or `andThen` for each type that is a monad, the Lean standard library contains a type class that allow them to be overloaded, so that the same operators can be used for *any* monad. Monads have two operations, which are the equivalent of `ok` and `andThen`:

```
class Monad (m : Type → Type) where
  pure : α → m α
  bind : m α → (α → m β) → m β
```

This definition is slightly simplified. The actual definition in the Lean library is somewhat more involved, and will be presented later.

The `Monad` instances for `Option` and `Except` can be created by adapting the definitions of their respective `andThen` operations:

```
instance : Monad Option where
  pure x := some x
  bind opt next :=
    match opt with
    | none => none
    | some x => next x

instance : Monad (Except ε) where
  pure x := Except.ok x
  bind attempt next :=
    match attempt with
    | Except.error e => Except.error e
    | Except.ok x => next x
```

As an example, `firstThirdFifthSeventh` was defined separately for `Option α` and `Except String α` return types. Now, it can be defined polymorphically for *any* monad. It does, however, require a lookup function as an argument, because different monads might fail to find a result in different ways. The infix version of `bind` is `>>=`, which plays the same role as `~~>` in the examples.

```
def firstThirdFifthSeventh [Monad m] (lookup : List α → Nat → m α) (xs : List α)
: m (α × α × α × α) :=
  lookup xs 0 >>= fun first =>
  lookup xs 2 >>= fun third =>
  lookup xs 4 >>= fun fifth =>
  lookup xs 6 >>= fun seventh =>
  pure (first, third, fifth, seventh)
```

Given example lists of slow mammals and fast birds, this implementation of `firstThirdFifthSeventh` can be used with `Option`:

```
def slowMammals : List String :=
  ["Three-toed sloth", "Slow loris"]

def fastBirds : List String := [
  "Peregrine falcon",
  "Saker falcon",
  "Golden eagle",
  "Gray-headed albatross",
  "Spur-winged goose",
  "Swift",
  "Anna's hummingbird"
]
```

```
#eval firstThirdFifthSeventh (fun xs i => xs[i]?) slowMammals
```

```
none
```

```
#eval firstThirdFifthSeventh (fun xs i => xs[i]?) fastBirds
```

```
some ("Peregrine falcon", "Golden eagle", "Spur-winged goose", "Anna's
hummingbird")
```

After renaming `Except`'s lookup function `get` to something more specific, the very same implementation of `firstThirdFifthSeventh` can be used with `Except` as well:

```
def getOrExcept (xs : List α) (i : Nat) : Except String α :=
  match xs[i]? with
  | none => Except.error s!"Index {i} not found (maximum is {xs.length - 1})"
  | some x => Except.ok x
```

```
#eval firstThirdFifthSeventh getOrExcept slowMammals
```

```
Except.error "Index 2 not found (maximum is 1)"
```

```
#eval firstThirdFifthSeventh getOrExcept fastBirds
```

```
Except.ok ("Peregrine falcon", "Golden eagle", "Spur-winged goose", "Anna's
hummingbird")
```

The fact that `m` must have a `Monad` instance means that the `>>=` and `pure` operations are available.

General Monad Operations

Because many different types are monads, functions that are polymorphic over *any* monad are very powerful. For example, the function `mapM` is a version of `map` that uses a `Monad` to sequence and combine the results of applying a function:

```
def mapM [Monad m] (f : α → m β) : List α → m (List β)
  | [] => pure []
  | x :: xs =>
    f x >>= fun hd =>
    mapM f xs >>= fun tl =>
    pure (hd :: tl)
```

The return type of the function argument `f` determines which `Monad` instance will be used. In other words, `mapM` can be used for functions that produce logs, for functions that can fail, or for functions that use mutable state. Because `f`'s type determines the available effects, they can be tightly controlled by API designers.

As described in this chapter's introduction, `State σ α` represents programs that make use of a mutable variable of type `σ` and return a value of type `α`. These programs are actually functions from a starting state to a pair of a value and a final state. The `Monad` class requires that its parameter expect a single type argument—that is, it should be a `Type → Type`. This means that the instance for `State` should mention the state type `σ`, which becomes a parameter to the instance:

```
instance : Monad (State σ) where
  pure x := fun s => (s, x)
  bind first next :=
    fun s =>
      let (s', x) := first s
      next x s'
```

This means that the type of the state cannot change between calls to `get` and `set` that are sequenced using `bind`, which is a reasonable rule for stateful computations. The operator `increment` increases a saved state by a given amount, returning the old value:

```
def increment (howMuch : Int) : State Int Int :=
  get >>= fun i =>
  set (i + howMuch) >>= fun () =>
  pure i
```

Using `mapM` with `increment` results in a program that computes the sum of the entries in a list. More specifically, the mutable variable contains the sum so far, while the resulting list contains a running sum. In other words, `mapM increment` has type `List Int → State Int`

`(List Int)`, and expanding the definition of `State` yields `List Int → Int → (Int × List Int)`. It takes an initial sum as an argument, which should be `0`:

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

```
(15, [0, 1, 3, 6, 10])
```

A logging effect can be represented using `WithLog`. Just like `State`, its `Monad` instance is polymorphic with respect to the type of the logged data:

```
instance : Monad (WithLog logged) where
  pure x := {log := [], val := x}
  bind result next :=
    let {log := thisOut, val := thisRes} := result
    let {log := nextOut, val := nextRes} := next thisRes
    {log := thisOut ++ nextOut, val := nextRes}
```

`saveIfEven` is a function that logs even numbers but returns its argument unchanged:

```
def saveIfEven (i : Int) : WithLog Int Int :=
  (if isEven i then
    save i
   else pure ()) >>= fun () =>
  pure i
```

Using this function with `mapM` results in a log containing even numbers paired with an unchanged input list:

```
#eval mapM saveIfEven [1, 2, 3, 4, 5]
```

```
{ log := [2, 4], val := [1, 2, 3, 4, 5] }
```

The Identity Monad

Monads encode programs with effects, such as failure, exceptions, or logging, into explicit representations as data and functions. Sometimes, however, an API will be written to use a monad for flexibility, but the API's client may not require any encoded effects. The *identity monad* is a monad that has no effects. It allows pure code to be used with monadic APIs:

```
def Id (t : Type) : Type := t

instance : Monad Id where
  pure x := x
  bind x f := f x
```

The type of `pure` should be `α → Id α`, but `Id α` reduces to just `α`. Similarly, the type of `bind` should be `α → (α → Id β) → Id β`. Because this reduces to `α → (α → β) → β`, the second argument can be applied to the first to find the result.

With the identity monad, `mapM` becomes equivalent to `map`. To call it this way, however, Lean requires a hint that the intended monad is `Id`:

```
#eval mapM (m := Id) (· + 1) [1, 2, 3, 4, 5]
```

```
[2, 3, 4, 5, 6]
```

Omitting the hint results in an error:

```
#eval mapM (· + 1) [1, 2, 3, 4, 5]
```

```
failed to synthesize
  HAdd Nat Nat (?m.4319 ?m.4321)

Additional diagnostic information may be available using the `set_option
diagnostics true` command.
```

In this error, the application of one metavariable to another indicates that Lean doesn't run the type-level computation backwards. The return type of the function is expected to be the monad applied to some other type. Similarly, using `mapM` with a function whose type doesn't provide any specific hints about which monad is to be used results in an "instance problem stuck" message:

```
-- TODO added type annotation - check
#eval mapM (fun (x : Nat) => x) [1, 2, 3, 4, 5]
```

```
typeclass instance problem is stuck, it is often due to metavariables
  Monad ?m.4319
```

The Monad Contract

Just as every pair of instances of `BEq` and `Hashable` should ensure that any two equal values have the same hash, there is a contract that each instance of `Monad` should obey. First, `pure` should be a left identity of `bind`. That is, `bind (pure v) f` should be the same as `f v`. Secondly, `pure` should be a right identity of `bind`, so `bind v pure` is the same as `v`. Finally, `bind` should be associative, so `bind (bind v f) g` is the same as `bind v (fun x => bind (f x) g)`.

This contract specifies the expected properties of programs with effects more generally. Because `pure` has no effects, sequencing its effects with `bind` shouldn't change the result. The associative property of `bind` basically says that the sequencing bookkeeping itself doesn't matter, so long as the order in which things are happening is preserved.

Exercises

Mapping on a Tree

Define a function `BinTree.mapM`. By analogy to `mapM` for lists, this function should apply a monadic function to each data entry in a tree, as a preorder traversal. The type signature should be:

```
def BinTree.mapM [Monad m] (f : α → m β) : BinTree α → m (BinTree β)
```

The Option Monad Contract

First, write a convincing argument that the `Monad` instance for `Option` satisfies the monad contract. Then, consider the following instance:

```
instance : Monad Option where
  pure x := some x
  bind opt next := none
```

Both methods have the correct type. Why does this instance violate the monad contract?

Example: Arithmetic in Monads

Monads are a way of encoding programs with side effects into a language that does not have them. It would be easy to read this as a sort of admission that pure functional programs are missing something important, requiring programmers to jump through hoops just to write a normal program. However, while using the `Monad` API does impose a syntactic cost on a program, it brings two important benefits:

1. Programs must be honest about which effects they use in their types. A quick glance at a type signature describes *everything* that the program can do, rather than just what it accepts and what it returns.
2. Not every language provides the same effects. For example, only some languages have exceptions. Other languages have unique, exotic effects, such as Icon's searching over multiple values and Scheme or Ruby's continuations. Because monads can encode *any* effect, programmers can choose which ones are the best fit for a given application, rather than being stuck with what the language developers provided.

One example of a program that can make sense in a variety of monads is an evaluator for arithmetic expressions.

Arithmetic Expressions

An arithmetic expression is either a literal integer or a primitive binary operator applied to two expressions. The operators are addition, subtraction, multiplication, and division:

```
inductive Expr (op : Type) where
  | const : Int → Expr op
  | prim : op → Expr op → Expr op → Expr op


inductive Arith where
  | plus
  | minus
  | times
  | div
```

The expression `2 + 3` is represented:

```
open Expr in
open Arith in
def twoPlusThree : Expr Arith :=
  prim plus (const 2) (const 3)
```

and `14 / (45 - 5 * 9)` is represented:

```
open Expr in
open Arith in
def fourteenDivided : Expr Arith :=
  prim div (const 14) (prim minus (const 45) (prim times (const 5) (const 9)))
```

Evaluating Expressions

Because expressions include division, and division by zero is undefined, evaluation might fail. One way to represent failure is to use `Option`:

```
def evaluateOption : Expr Arith → Option Int
  | Expr.const i => pure i
  | Expr.prim p e1 e2 =>
    evaluateOption e1 >>= fun v1 =>
    evaluateOption e2 >>= fun v2 =>
    match p with
    | Arith.plus => pure (v1 + v2)
    | Arith.minus => pure (v1 - v2)
    | Arith.times => pure (v1 * v2)
    | Arith.div => if v2 == 0 then none else pure (v1 / v2)
```

This definition uses the `Monad Option` instance to propagate failures from evaluating both branches of a binary operator. However, the function mixes two concerns: evaluating subexpressions and applying a binary operator to the results. It can be improved by splitting it into two functions:

```
def applyPrim : Arith → Int → Int → Option Int
  | Arith.plus, x, y => pure (x + y)
  | Arith.minus, x, y => pure (x - y)
  | Arith.times, x, y => pure (x * y)
  | Arith.div, x, y => if y == 0 then none else pure (x / y)

def evaluateOption : Expr Arith → Option Int
  | Expr.const i => pure i
  | Expr.prim p e1 e2 =>
    evaluateOption e1 >>= fun v1 =>
    evaluateOption e2 >>= fun v2 =>
    applyPrim p v1 v2
```

Running `#eval evaluateOption fourteenDivided` yields `none`, as expected, but this is not a very useful error message. Because the code was written using `>>=` rather than by explicitly handling the `none` constructor, only a small modification is required for it to provide an error message on failure:

```
def applyPrim : Arith → Int → Int → Except String Int
  | Arith.plus, x, y => pure (x + y)
  | Arith.minus, x, y => pure (x - y)
  | Arith.times, x, y => pure (x * y)
  | Arith.div, x, y =>
    if y == 0 then
      Except.error s!"Tried to divide {x} by zero"
    else pure (x / y)


def evaluateExcept : Expr Arith → Except String Int
  | Expr.const i => pure i
  | Expr.prim p e1 e2 =>
    evaluateExcept e1 >>= fun v1 =>
    evaluateExcept e2 >>= fun v2 =>
    applyPrim p v1 v2
```

The only difference is that the type signature mentions `Except String` instead of `Option`, and the failing case uses `Except.error` instead of `none`. By making `evaluate` polymorphic over its monad and passing it `applyPrim` as an argument, a single evaluator becomes capable of both forms of error reporting:

```
def applyPrimOption : Arith → Int → Int → Option Int
  | Arith.plus, x, y => pure (x + y)
  | Arith.minus, x, y => pure (x - y)
  | Arith.times, x, y => pure (x * y)
  | Arith.div, x, y =>
    if y == 0 then
      none
    else pure (x / y)

def applyPrimExcept : Arith → Int → Int → Except String Int
  | Arith.plus, x, y => pure (x + y)
  | Arith.minus, x, y => pure (x - y)
  | Arith.times, x, y => pure (x * y)
  | Arith.div, x, y =>
    if y == 0 then
      Except.error s!"Tried to divide {x} by zero"
    else pure (x / y)

def evaluateM [Monad m] (applyPrim : Arith → Int → Int → m Int): Expr Arith → m
Int
  | Expr.const i => pure i
  | Expr.prim p e1 e2 =>
    evaluateM applyPrim e1 >>= fun v1 =>
    evaluateM applyPrim e2 >>= fun v2 =>
    applyPrim p v1 v2
```

Using it with `applyPrimOption` works just like the first version of `evaluate`:

```
#eval evaluateM applyPrimOption fourteenDivided
```

```
none
```

Similarly, using it with `applyPrimExcept` works just like the version with error messages:

```
#eval evaluateM applyPrimExcept fourteenDivided
```

```
Except.error "Tried to divide 14 by zero"
```

The code can still be improved. The functions `applyPrimOption` and `applyPrimExcept` differ only in their treatment of division, which can be extracted into another parameter to the evaluator:

```
def applyDivOption (x : Int) (y : Int) : Option Int :=
    if y == 0 then
      none
    else pure (x / y)

def applyDivExcept (x : Int) (y : Int) : Except String Int :=
    if y == 0 then
      Except.error s!"Tried to divide {x} by zero"
    else pure (x / y)

def applyPrim [Monad m] (applyDiv : Int → Int → m Int) : Arith → Int → Int → m
Int
  | Arith.plus, x, y => pure (x + y)
  | Arith.minus, x, y => pure (x - y)
  | Arith.times, x, y => pure (x * y)
  | Arith.div, x, y => applyDiv x y

def evaluateM [Monad m] (applyDiv : Int → Int → m Int): Expr Arith → m Int
  | Expr.const i => pure i
  | Expr.prim p e1 e2 =>
    evaluateM applyDiv e1 >>= fun v1 =>
    evaluateM applyDiv e2 >>= fun v2 =>
    applyPrim applyDiv p v1 v2
```

In this refactored code, the fact that the two code paths differ only in their treatment of failure has been made fully apparent.

Further Effects

Failure and exceptions are not the only kinds of effects that can be interesting when working with an evaluator. While division's only side effect is failure, adding other primitive operators to the expressions make it possible to express other effects.

The first step is an additional refactoring, extracting division from the datatype of primitives:

```
inductive Prim (special : Type) where
  | plus
  | minus
  | times
  | other : special → Prim special

inductive CanFail where
  | div
```

The name `CanFail` suggests that the effect introduced by division is potential failure.

The second step is to broaden the scope of the division handler argument to `evaluateM` so that it can process any special operator:

```
def divOption : CanFail → Int → Int → Option Int
  | CanFail.div, x, y => if y == 0 then none else pure (x / y)

def divExcept : CanFail → Int → Int → Except String Int
  | CanFail.div, x, y =>
    if y == 0 then
      Except.error s!"Tried to divide {x} by zero"
    else pure (x / y)

def applyPrim [Monad m] (applySpecial : special → Int → Int → m Int) : Prim
special → Int → Int → m Int
  | Prim.plus, x, y => pure (x + y)
  | Prim.minus, x, y => pure (x - y)
  | Prim.times, x, y => pure (x * y)
  | Prim.other op, x, y => applySpecial op x y

def evaluateM [Monad m] (applySpecial : special → Int → Int → m Int): Expr (Prim
special) → m Int
  | Expr.const i => pure i
  | Expr.prim p e1 e2 =>
    evaluateM applySpecial e1 >>= fun v1 =>
    evaluateM applySpecial e2 >>= fun v2 =>
    applyPrim applySpecial p v1 v2
```

No Effects

The type `Empty` has no constructors, and thus no values, like the `Nothing` type in Scala or Kotlin. In Scala and Kotlin, `Nothing` can represent computations that never return a result, such as functions that crash the program, throw exceptions, or always fall into infinite loops. An argument to a function or method of type `Nothing` indicates dead code, as there will never be a suitable argument value. Lean doesn't support infinite loops and exceptions, but `Empty` is still useful as an indication to the type system that a function cannot be called. Using the syntax `nomatch E` when `E` is an expression whose type has no constructors indicates to Lean that the current expression need not return a result, because it could never have been called.

Using `Empty` as the parameter to `Prim` indicates that there are no additional cases beyond `Prim.plus`, `Prim.minus`, and `Prim.times`, because it is impossible to come up with a value of type `Empty` to place in the `Prim.other` constructor. Because a function to apply an operator of type `Empty` to two integers can never be called, it doesn't need to return a result. Thus, it can be used in *any* monad:

```
def applyEmpty [Monad m] (op : Empty) (_ : Int) (_ : Int) : m Int :=
  nomatch op
```

This can be used together with `Id`, the identity monad, to evaluate expressions that have no effects whatsoever:

```
open Expr Prim in
#eval evaluateM (m := Id) applyEmpty (prim plus (const 5) (const (-14)))
```

```
-9
```

Nondeterministic Search

Instead of simply failing when encountering division by zero, it would also be sensible to backtrack and try a different input. Given the right monad, the very same `evaluateM` can perform a nondeterministic search for a *set* of answers that do not result in failure. This requires, in addition to division, some means of specifying a choice of results. One way to do this is to add a function `choose` to the language of expressions that instructs the evaluator to pick either of its arguments while searching for non-failing results.

The result of the evaluator is now a multiset of values, rather than a single value. The rules for evaluation into a multiset are:

- Constants n evaluate to singleton sets n.
- Arithmetic operators other than division are called on each pair from the Cartesian product of the operators, so $X + Y$ evaluates to $\{x + y \mid x \in X, y \in Y\}$.
- Division X / Y evaluates to $\{x / y \mid x \in X, y \in Y, y \neq 0\}$. In other words, all 0 values in Y are thrown out.
- A choice $\mathrm{choose}(x, y)$ evaluates to $\{x, y\}$.

For example, $1 + \mathrm{choose}(2, 5)$ evaluates to $\{3, 6\}$, $1 + 2/0$ evaluates to $\{\}$, and $90/(\mathrm{choose}(-5, 5) + 5)$ evaluates to $\{9\}$. Using multisets instead of true sets simplifies the code by removing the need to check for uniqueness of elements.

A monad that represents this non-deterministic effect must be able to represent a situation in which there are no answers, and a situation in which there is at least one answer together with any remaining answers:

```
inductive Many (α : Type) where
  | none : Many α
  | more : α → (Unit → Many α) → Many α
```

This datatype looks very much like `List`. The difference is that where `cons` stores the rest of the list, `more` stores a function that should compute the next value on demand. This means that a consumer of `Many` can stop the search when some number of results have been found.

A single result is represented by a `more` constructor that returns no further results:

```
def Many.one (x : α) : Many α := Many.more x (fun () => Many.none)
```

The union of two multisets of results can be computed by checking whether the first multiset is empty. If so, the second multiset is the union. If not, the union consists of the first element of the first multiset followed by the union of the rest of the first multiset with the second multiset:

```
def Many.union : Many α → Many α → Many α
  | Many.none, ys => ys
  | Many.more x xs, ys => Many.more x (fun () => union (xs ()) ys)
```

It can be convenient to start a search process with a list of values. `Many.fromList` converts a list into a multiset of results:

```
def Many.fromList : List α → Many α
  | [] => Many.none
  | x :: xs => Many.more x (fun () => fromList xs)
```

Similarly, once a search has been specified, it can be convenient to extract either a number of values, or all the values:

```
def Many.take : Nat → Many α → List α
  | 0, _ => []
  | _ + 1, Many.none => []
  | n + 1, Many.more x xs => x :: (xs ()).take n

def Many.takeAll : Many α → List α
  | Many.none => []
  | Many.more x xs => x :: (xs ()).takeAll
```

A `Monad Many` instance requires a `bind` operator. In a nondeterministic search, sequencing two operations consists of taking all possibilities from the first step and running the rest of the program on each of them, taking the union of the results. In other words, if the first step returns three possible answers, the second step needs to be tried for all three. Because the second step can return any number of answers for each input, taking their union represents the entire search space.

```
def Many.bind : Many α → (α → Many β) → Many β
  | Many.none, _ =>
    Many.none
  | Many.more x xs, f =>
    (f x).union (bind (xs ()) f)
```

`Many.one` and `Many.bind` obey the monad contract. To check that `Many.bind (Many.one v) f` is the same as `f v`, start by evaluating the expression as far as possible:

```
Many.bind (Many.one v) f
===>
Many.bind (Many.more v (fun () => Many.none)) f
===>
(f v).union (Many.bind Many.none f)
===>
(f v).union Many.none
```

The empty multiset is a right identity of `union`, so the answer is equivalent to `f v`. To check that `Many.bind v Many.one` is the same as `v`, consider that `bind` takes the union of applying `Many.one` to each element of `v`. In other words, if `v` has the form `{v1, v2, v3, ..., vn}`, then `Many.bind v Many.one` is `{v1} ∪ {v2} ∪ {v3} ∪ ... ∪ {vn}`, which is `{v1, v2, v3, ..., vn}`.

Finally, to check that `Many.bind` is associative, check that `Many.bind (Many.bind bind v f) g` is the same as `Many.bind v (fun x => Many.bind (f x) g)`. If `v` has the form `{v1, v2, v3, ..., vn}`, then:

```
Many.bind v f
===>
f v1 ∪ f v2 ∪ f v3 ∪ ... ∪ f vn
```

which means that

```
Many.bind (Many.bind bind v f) g
===>
Many.bind (f v1) g ∪
Many.bind (f v2) g ∪
Many.bind (f v3) g ∪
... ∪
Many.bind (f vn) g
```

Similarly,

```
Many.bind v (fun x => Many.bind (f x) g)
===>
(fun x => Many.bind (f x) g) v1 ∪
(fun x => Many.bind (f x) g) v2 ∪
(fun x => Many.bind (f x) g) v3 ∪
... ∪
(fun x => Many.bind (f x) g) vn
===>
Many.bind (f v1) g ∪
Many.bind (f v2) g ∪
Many.bind (f v3) g ∪
... ∪
Many.bind (f vn) g
```

Thus, both sides are equal, so `Many.bind` is associative.

The resulting monad instance is:

```
instance : Monad Many where
  pure := Many.one
  bind := Many.bind
```

An example search using this monad finds all the combinations of numbers in a list that add to 15:

```
def addsTo (goal : Nat) : List Nat → Many (List Nat)
  | [] =>
    if goal == 0 then
      pure []
    else
      Many.none
  | x :: xs =>
    if x > goal then
      addsTo goal xs
    else
      (addsTo goal xs).union
        (addsTo (goal - x) xs >>= fun answer =>
         pure (x :: answer))
```

The search process is recursive over the list. The empty list is a successful search when the goal is `0`; otherwise, it fails. When the list is non-empty, there are two possibilities: either the head of the list is greater than the goal, in which case it cannot participate in any successful searches, or it is not, in which case it can. If the head of the list is *not* a candidate, then the search proceeds to the tail of the list. If the head is a candidate, then there are two possibilities to be combined with `Many.union`: either the solutions found contain the head, or they do not. The solutions that do not contain the head are found with a recursive call on the tail, while the solutions that do contain it result from subtracting the head from the goal, and then attaching the head to the solutions that result from the recursive call.

The helper `printList` ensures that one result is displayed per line:

```
def printList [ToString α] : List α → IO Unit
  | [] => pure ()
  | x :: xs => do
    IO.println x
    printList xs

#eval printList (addsTo 15 [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]).takeAll
```

```
[7, 8]
[6, 9]
[5, 10]
[4, 5, 6]
[3, 5, 7]
[3, 4, 8]
[2, 6, 7]
[2, 5, 8]
[2, 4, 9]
[2, 3, 10]
[2, 3, 4, 6]
[1, 6, 8]
[1, 5, 9]
[1, 4, 10]
[1, 3, 5, 6]
[1, 3, 4, 7]
[1, 2, 5, 7]
[1, 2, 4, 8]
[1, 2, 3, 9]
[1, 2, 3, 4, 5]
```

Returning to the arithmetic evaluator that produces multisets of results, the `both` and `neither` operators can be written as follows:

```
inductive NeedsSearch
  | div
  | choose

def applySearch : NeedsSearch → Int → Int → Many Int
  | NeedsSearch.choose, x, y =>
    Many.fromList [x, y]
  | NeedsSearch.div, x, y =>
    if y == 0 then
      Many.none
    else Many.one (x / y)
```

Using these operators, the earlier examples can be evaluated:

```
open Expr Prim NeedsSearch

#eval (evaluateM applySearch (prim plus (const 1) (prim (other choose) (const 2)
(const 5)))).takeAll
```

```
[3, 6]
```

```
#eval (evaluateM applySearch (prim plus (const 1) (prim (other div) (const 2)
(const 0)))).takeAll
```

```
[]
```

```
#eval (evaluateM applySearch (prim (other div) (const 90) (prim plus (prim
(other choose) (const (-5)) (const 5)) (const 5)))).takeAll
```

```
[9]
```

Custom Environments

The evaluator can be made user-extensible by allowing strings to be used as operators, and then providing a mapping from strings to a function that implements them. For example, users could extend the evaluator with a remainder operator or with one that returns the maximum of its two arguments. The mapping from function names to function implementations is called an *environment*.

The environments needs to be passed in each recursive call. Initially, it might seem that `evaluateM` needs an extra argument to hold the environment, and that this argument should be passed to each recursive invocation. However, passing an argument like this is another form of monad, so an appropriate `Monad` instance allows the evaluator to be used unchanged.

Using functions as a monad is typically called a *reader* monad. When evaluating expressions in the reader monad, the following rules are used:

- Constants n evaluate to constant functions $\lambda e. n$,
- Arithmetic operators evaluate to functions that pass their arguments on, so $f + g$ evaluates to $\lambda e. f(e) + g(e)$, and
- Custom operators evaluate to the result of applying the custom operator to the arguments, so $f \ \mathrm{OP} \ g$ evaluates to

  $$\lambda e. \begin{cases} h(f(e), g(e)) & \text{if } e \text{ contains } (\mathrm{OP}, h) \\ 0 & \text{otherwise} \end{cases}$$

  with 0 serving as a fallback in case an unknown operator is applied.

To define the reader monad in Lean, the first step is to define the `Reader` type and the effect that allows users to get ahold of the environment:

```
def Reader (ρ : Type) (α : Type) : Type := ρ → α
```

```
def read : Reader ρ ρ := fun env => env
```

By convention, the Greek letter `ρ`, which is pronounced "rho", is used for environments.

The fact that constants in arithmetic expressions evaluate to constant functions suggests that the appropriate definition of `pure` for `Reader` is a a constant function:

```
def Reader.pure (x : α) : Reader ρ α := fun _ => x
```

On the other hand, `bind` is a bit tricker. Its type is `Reader ρ α → (α → Reader ρ β) → Reader ρ β`. This type can be easier to understand by unfolding the definition of `Reader`, which yields `(ρ → α) → (α → ρ → β) → ρ → β`. It should take an environment-accepting function as its first argument, while the second argument should transform the result of the environment-accepting function into yet another environment-accepting function. The result of combining these is itself a function, waiting for an environment.

It's possible to use Lean interactively to get help writing this function. The first step is to write down the arguments and return type, being very explicit in order to get as much help as possible, with an underscore for the definition's body:

```
def Reader.bind {ρ : Type} {α : Type} {β : Type}
  (result : ρ → α) (next : α → ρ → β) : ρ → β :=
  _
```

Lean provides a message that describes which variables are available in scope, and the type that's expected for the result. The `⊢` symbol, called a *turnstile* due to its resemblance to subway entrances, separates the local variables from the desired type, which is `ρ → β` in this message:

```
don't know how to synthesize placeholder
context:
ρ α β : Type
result : ρ → α
next : α → ρ → β
⊢ ρ → β
```

Because the return type is a function, a good first step is to wrap a `fun` around the underscore:

```
def Reader.bind {ρ : Type} {α : Type} {β : Type}
  (result : ρ → α) (next : α → ρ → β) : ρ → β :=
  fun env => _
```

The resulting message now shows the function's argument as a local variable:

```
don't know how to synthesize placeholder
context:
ρ α β : Type
result : ρ → α
next : α → ρ → β
env : ρ
⊢ β
```

The only thing in the context that can produce a `β` is `next`, and it will require two arguments to do so. Each argument can itself be an underscore:

```
def Reader.bind {ρ : Type} {α : Type} {β : Type}
  (result : ρ → α) (next : α → ρ → β) : ρ → β :=
  fun env => next _ _
```

The two underscores have the following respective messages associated with them:

```
don't know how to synthesize placeholder
context:
ρ α β : Type
result : ρ → α
next : α → ρ → β
env : ρ
⊢ α
```

```
don't know how to synthesize placeholder
context:
ρ α β : Type
result : ρ → α
next : α → ρ → β
env : ρ
⊢ ρ
```

Attacking the first underscore, only one thing in the context can produce an `α`, namely `result`:

```
def Reader.bind {ρ : Type} {α : Type} {β : Type}
  (result : ρ → α) (next : α → ρ → β) : ρ → β :=
  fun env => next (result _) _
```

Now, both underscores have the same error message:

```
don't know how to synthesize placeholder
context:
ρ α β : Type
result : ρ → α
next : α → ρ → β
env : ρ
⊢ ρ
```

Happily, both underscores can be replaced by `env`, yielding:

```
def Reader.bind {ρ : Type} {α : Type} {β : Type}
  (result : ρ → α) (next : α → ρ → β) : ρ → β :=
  fun env => next (result env) env
```

The final version can be obtained by undoing the unfolding of `Reader` and cleaning up the explicit details:

```
def Reader.bind (result : Reader ρ α) (next : α → Reader ρ β) : Reader ρ β :=
  fun env => next (result env) env
```

It's not always possible to write correct functions by simply "following the types", and it carries the risk of not understanding the resulting program. However, it can also be easier to understand a program that has been written than one that has not, and the process of filling in the underscores can bring insights. In this case, `Reader.bind` works just like `bind` for `Id`, except it accepts an additional argument that it then passes down to its arguments, and this intuition can help in understanding how it works.

`Reader.pure`, which generates constant functions, and `Reader.bind` obey the monad contract. To check that `Reader.bind (Reader.pure v) f` is the same as `f v`, it's enough to replace definitions until the last step:

```
Reader.bind (Reader.pure v) f
===>
fun env => f ((Reader.pure v) env) env
===>
fun env => f ((fun _ => v) env) env
===>
fun env => f v env
===>
f v
```

For every function `f`, `fun x => f x` is the same as `f`, so the first part of the contract is satisfied. To check that `Reader.bind r Reader.pure` is the same as `r`, a similar technique works:

```
Reader.bind r Reader.pure
===>
fun env => Reader.pure (r env) env
===>
fun env => (fun _ => (r env)) env
===>
fun env => r env
```

Because reader actions `r` are themselves functions, this is the same as `r`. To check associativity, the same thing can be done for both `Reader.bind (Reader.bind r f) g` and `Reader.bind r (fun x => Reader.bind (f x) g)`:

```
Reader.bind (Reader.bind r f) g
===>
fun env => g ((Reader.bind r f) env) env
===>
fun env => g ((fun env' => f (r env') env') env) env
===>
fun env => g (f (r env) env) env
```

```
Reader.bind r (fun x => Reader.bind (f x) g)
===>
Reader.bind r (fun x => fun env => g (f x env) env)
===>
fun env => (fun x => fun env' => g (f x env') env') (r env) env
===>
fun env => (fun env' => g (f (r env) env') env') env
===>
fun env => g (f (r env) env) env
```

Thus, a `Monad (Reader ρ)` instance is justified:

```
instance : Monad (Reader ρ) where
  pure x := fun _ => x
  bind x f := fun env => f (x env) env
```

The custom environments that will be passed to the expression evaluator can be represented as lists of pairs:

```
abbrev Env : Type := List (String × (Int → Int → Int))
```

For instance, `exampleEnv` contains maximum and modulus functions:

```
def exampleEnv : Env := [("max", max), ("mod", (· % ·))]
```

Lean already has a function `List.lookup` that finds the value associated with a key in a list of pairs, so `applyPrimReader` needs only check whether the custom function is present in the environment. It returns `0` if the function is unknown:

```
def applyPrimReader (op : String) (x : Int) (y : Int) : Reader Env Int :=
  read >>= fun env =>
  match env.lookup op with
  | none => pure 0
  | some f => pure (f x y)
```

Using `evaluateM` with `applyPrimReader` and an expression results in a function that expects an environment. Luckily, `exampleEnv` is available:

```
open Expr Prim in
#eval evaluateM applyPrimReader (prim (other "max") (prim plus (const 5) (const
4)) (prim times (const 3) (const 2))) exampleEnv
```

```
9
```

Like `Many` , `Reader` is an example of an effect that is difficult to encode in most languages, but type classes and monads make it just as convenient as any other effect. The dynamic or special variables found in Common Lisp, Clojure, and Emacs Lisp can be used like `Reader` . Similarly, Scheme and Racket's parameter objects are an effect that exactly correspond to `Reader` . The Kotlin idiom of context objects can solve a similar problem, but they are fundamentally a means of passing function arguments automatically, so this idiom is more like the encoding as a reader monad than it is an effect in the language.

Exercises

Checking Contracts

Check the monad contract for `State σ` and `Except ε` .

Readers with Failure

Adapt the reader monad example so that it can also indicate failure when the custom operator is not defined, rather than just returning zero. In other words, given these definitions:

```
def ReaderOption (ρ : Type) (α : Type) : Type := ρ → Option α

def ReaderExcept (ε : Type) (ρ : Type) (α : Type) : Type := ρ → Except ε α
```

do the following:

1. Write suitable `pure` and `bind` functions
2. Check that these functions satisfy the `Monad` contract
3. Write `Monad` instances for `ReaderOption` and `ReaderExcept`
4. Define suitable `applyPrim` operators and test them with `evaluateM` on some example expressions

A Tracing Evaluator

The `WithLog` type can be used with the evaluator to add optional tracing of some operations. In particular, the type `ToTrace` can serve as a signal to trace a given operator:

```
inductive ToTrace (α : Type) : Type where
  | trace : α → ToTrace α
```

For the tracing evaluator, expressions should have type `Expr (Prim (ToTrace (Prim Empty)))`. This says that the operators in the expression consist of addition, subtraction, and multiplication, augmented with traced versions of each. The innermost argument is `Empty` to signal that there are no further special operators inside of `trace`, only the three basic ones.

Do the following:

1. Implement a `Monad (WithLog logged)` instance
2. Write an `applyTraced` function to apply traced operators to their arguments, logging both the operator and the arguments, with type `ToTrace (Prim Empty) → Int → Int → WithLog (Prim Empty × Int × Int) Int`

If the exercise has been completed correctly, then

```
open Expr Prim ToTrace in
#eval evaluateM applyTraced (prim (other (trace times)) (prim (other (trace
plus)) (const 1) (const 2)) (prim (other (trace minus)) (const 3) (const 4)))
```

should result in

```
{ log := [(Prim.plus, 1, 2), (Prim.minus, 3, 4), (Prim.times, 3, -1)], val := -3
}
```

Hint: values of type `Prim Empty` will appear in the resulting log. In order to display them as a result of `#eval`, the following instances are required:

```
deriving instance Repr for WithLog
deriving instance Repr for Empty
deriving instance Repr for Prim
```

do-Notation for Monads

While APIs based on monads are very powerful, the explicit use of `>>=` with anonymous functions is still somewhat noisy. Just as infix operators are used instead of explicit calls to `HAdd.hAdd`, Lean provides a syntax for monads called *`do`-notation* that can make programs that use monads easier to read and write. This is the very same `do`-notation that is used to write programs in `IO`, and `IO` is also a monad.

In Hello, World!, the `do` syntax is used to combine `IO` actions, but the meaning of these programs is explained directly. Understanding how to program with monads means that `do` can now be explained in terms of how it translates into uses of the underlying monad operators.

The first translation of `do` is used when the only statement in the `do` is a single expression `E`. In this case, the `do` is removed, so

```
do E
```

translates to

```
E
```

The second translation is used when the first statement of the `do` is a `let` with an arrow, binding a local variable. This translates to a use of `>>=` together with a function that binds that very same variable, so

```
do let x ← E1
   Stmt
   ...
   En
```

translates to

```
E1 >>= fun x =>
do Stmt
   ...
   En
```

When the first statement of the `do` block is an expression, then it is considered to be a monadic action that returns `Unit`, so the function matches the `Unit` constructor and

```
do E1
   Stmt
   ...
   En
```

translates to

```
E1 >>= fun () =>
do Stmt
   ...
   En
```

Finally, when the first statement of the `do` block is a `let` that uses `:=`, the translated form is an ordinary let expression, so

```
do let x := E1
   Stmt
   ...
   En
```

translates to

```
let x := E1
do Stmt
   ...
   En
```

The definition of `firstThirdFifthSeventh` that uses the `Monad` class looks like this:

```
def firstThirdFifthSeventh [Monad m] (lookup : List α → Nat → m α) (xs : List α)
: m (α × α × α × α) :=
  lookup xs 0 >>= fun first =>
  lookup xs 2 >>= fun third =>
  lookup xs 4 >>= fun fifth =>
  lookup xs 6 >>= fun seventh =>
  pure (first, third, fifth, seventh)
```

Using `do`-notation, it becomes significantly more readable:

```
def firstThirdFifthSeventh [Monad m] (lookup : List α → Nat → m α) (xs : List α)
: m (α × α × α × α) := do
  let first ← lookup xs 0
  let third ← lookup xs 2
  let fifth ← lookup xs 4
  let seventh ← lookup xs 6
  pure (first, third, fifth, seventh)
```

Without the `Monad` type class, the function `number` that numbers the nodes of a tree was written:

```
def number (t : BinTree α) : BinTree (Nat × α) :=
  let rec helper : BinTree α → State Nat (BinTree (Nat × α))
    | BinTree.leaf => ok BinTree.leaf
    | BinTree.branch left x right =>
      helper left ~~> fun numberedLeft =>
      get ~~> fun n =>
      set (n + 1) ~~> fun () =>
      helper right ~~> fun numberedRight =>
      ok (BinTree.branch numberedLeft (n, x) numberedRight)
  (helper t 0).snd
```

With `Monad` and `do`, its definition is much less noisy:

```
def number (t : BinTree α) : BinTree (Nat × α) :=
  let rec helper : BinTree α → State Nat (BinTree (Nat × α))
    | BinTree.leaf => pure BinTree.leaf
    | BinTree.branch left x right => do
      let numberedLeft ← helper left
      let n ← get
      set (n + 1)
      let numberedRight ← helper right
      ok (BinTree.branch numberedLeft (n, x) numberedRight)
  (helper t 0).snd
```

All of the conveniences from `do` with `IO` are also available when using it with other monads. For example, nested actions also work in any monad. The original definition of `mapM` was:

```
def mapM [Monad m] (f : α → m β) : List α → m (List β)
  | [] => pure []
  | x :: xs =>
    f x >>= fun hd =>
    mapM f xs >>= fun tl =>
    pure (hd :: tl)
```

With `do`-notation, it can be written:

```
def mapM [Monad m] (f : α → m β) : List α → m (List β)
  | [] => pure []
  | x :: xs => do
    let hd ← f x
    let tl ← mapM f xs
    pure (hd :: tl)
```

Using nested actions makes it almost as short as the original non-monadic `map`:

```
def mapM [Monad m] (f : α → m β) : List α → m (List β)
  | [] => pure []
  | x :: xs => do
    pure ((← f x) :: (← mapM f xs))
```

Using nested actions, `number` can be made much more concise:

```
def increment : State Nat Nat := do
  let n ← get
  set (n + 1)
  pure n

def number (t : BinTree α) : BinTree (Nat × α) :=
  let rec helper : BinTree α → State Nat (BinTree (Nat × α))
    | BinTree.leaf => pure BinTree.leaf
    | BinTree.branch left x right => do
      pure (BinTree.branch (← helper left) ((← increment), x) (← helper right))
  (helper t 0).snd
```

Exercises

- Rewrite `evaluateM`, its helpers, and the different specific use cases using `do`-notation instead of explicit calls to `>>=`.
- Rewrite `firstThirdFifthSeventh` using nested actions.

The IO Monad

`IO` as a monad can be understood from two perspectives, which were described in the section on running programs. Each can help to understand the meanings of `pure` and `bind` for `IO`.

From the first perspective, an `IO` action is an instruction to Lean's run-time system. For example, the instruction might be "read a string from this file descriptor, then re-invoke the pure Lean code with the string". This perspective is an *exterior* one, viewing the program from the perspective of the operating system. In this case, `pure` is an `IO` action that does not request any effects from the RTS, and `bind` instructs the RTS to first carry out one potentially-effectful operation and then invoke the rest of the program with the resulting value.

From the second perspective, an `IO` action transforms the whole world. `IO` actions are actually pure, because they receive a unique world as an argument and then return the changed world. This perspective is an *interior* one that matches how `IO` is represented inside of Lean. The world is represented in Lean as a token, and the `IO` monad is structured to make sure that each token is used exactly once.

To see how this works, it can be helpful to peel back one definition at a time. The `#print` command reveals the internals of Lean datatypes and definitions. For example,

```
#print Nat
```

results in

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

and

```
#print Char.isAlpha
```

results in

```
def Char.isAlpha : Char → Bool :=
fun c => c.isUpper || c.isLower
```

Sometimes, the output of `#print` includes Lean features that have not yet been presented in this book. For example,

```
#print List.isEmpty
```

produces

```
def List.isEmpty.{u} : {α : Type u} → List α → Bool :=
fun {α} x =>
  match x with
  | [] => true
  | head :: tail => false
```

which includes a `.{u}` after the definition's name, and annotates types as `Type u` rather than just `Type`. This can be safely ignored for now.

Printing the definition of `IO` shows that it's defined in terms of simpler structures:

```
#print IO
```

```
@[reducible] def IO : Type → Type :=
EIO IO.Error
```

`IO.Error` represents all the errors that could be thrown by an `IO` action:

```
#print IO.Error
```

```
inductive IO.Error : Type
number of parameters: 0
constructors:
IO.Error.alreadyExists : Option String → UInt32 → String → IO.Error
IO.Error.otherError : UInt32 → String → IO.Error
IO.Error.resourceBusy : UInt32 → String → IO.Error
IO.Error.resourceVanished : UInt32 → String → IO.Error
IO.Error.unsupportedOperation : UInt32 → String → IO.Error
IO.Error.hardwareFault : UInt32 → String → IO.Error
IO.Error.unsatisfiedConstraints : UInt32 → String → IO.Error
IO.Error.illegalOperation : UInt32 → String → IO.Error
IO.Error.protocolError : UInt32 → String → IO.Error
IO.Error.timeExpired : UInt32 → String → IO.Error
IO.Error.interrupted : String → UInt32 → String → IO.Error
IO.Error.noFileOrDirectory : String → UInt32 → String → IO.Error
IO.Error.invalidArgument : Option String → UInt32 → String → IO.Error
IO.Error.permissionDenied : Option String → UInt32 → String → IO.Error
IO.Error.resourceExhausted : Option String → UInt32 → String → IO.Error
IO.Error.inappropriateType : Option String → UInt32 → String → IO.Error
IO.Error.noSuchThing : Option String → UInt32 → String → IO.Error
IO.Error.unexpectedEof : IO.Error
IO.Error.userError : String → IO.Error
```

`EIO ε α` represents `IO` actions that will either terminate with an error of type `ε` or succeed with a value of type `α`. This means that, like the `Except ε` monad, the `IO` monad includes the ability to define error handling and exceptions.

Peeling back another layer, `EIO` is itself defined in terms of a simpler structure:

```
#print EIO
```

```
def EIO : Type → Type → Type :=
fun ε => EStateM ε IO.RealWorld
```

The `EStateM` monad includes both errors and state—it's a combination of `Except` and `State`. It is defined using another type, `EStateM.Result`:

```
#print EStateM
```

```
def EStateM.{u} : Type u → Type u → Type u → Type u :=
fun ε σ α => σ → EStateM.Result ε σ α
```

In other words, a program with type `EStateM ε σ α` is a function that accepts an initial state of type `σ` and returns an `EStateM.Result ε σ α`.

`EStateM.Result` is very much like the definition of `Except`, with one constructor that indicates a successful termination and one constructor that indicates an error:

```
#print EStateM.Result
```

```
inductive EStateM.Result.{u} : Type u → Type u → Type u → Type u
number of parameters: 3
constructors:
EStateM.Result.ok : {ε σ α : Type u} → α → σ → EStateM.Result ε σ α
EStateM.Result.error : {ε σ α : Type u} → ε → σ → EStateM.Result ε σ α
```

Just like `Except ε α`, the `ok` constructor includes a result of type `α`, and the `error` constructor includes an exception of type `ε`. Unlike `Except`, both constructors have an additional state field that includes the final state of the computation.

The `Monad` instance for `EStateM ε σ` requires `pure` and `bind`. Just as with `State`, the implementation of `pure` for `EStateM` accepts an initial state and returns it unchanged, and just as with `Except`, it returns its argument in the `ok` constructor:

```
#print EStateM.pure
```

```
protected def EStateM.pure.{u} : {ε σ α : Type u} → α → EStateM ε σ α :=
fun {ε σ α} a s => EStateM.Result.ok a s
```

`protected` means that the full name `EStateM.pure` is needed even if the `EStateM` namespace has been opened.

Similarly, `bind` for `EStateM` takes an initial state as an argument. It passes this initial state to its first action. Like `bind` for `Except`, it then checks whether the result is an error. If so, the error is returned unchanged and the second argument to `bind` remains unused. If the result was a success, then the second argument is applied to both the returned value and to the resulting state.

```
#print EStateM.bind
```

```
protected def EStateM.bind.{u} : {ε σ α β : Type u} → EStateM ε σ α → (α →
EStateM ε σ β) → EStateM ε σ β :=
fun {ε σ α β} x f s =>
  match x s with
  | EStateM.Result.ok a s => f a s
  | EStateM.Result.error e s => EStateM.Result.error e s
```

Putting all of this together, `IO` is a monad that tracks state and errors at the same time. The collection of available errors is that given by the datatype `IO.Error`, which has constructors that describe many things that can go wrong in a program. The state is a type that represents the real world, called `IO.RealWorld`. Each basic `IO` action receives this real world and returns another one, paired either with an error or a result. In `IO`, `pure` returns the world unchanged, while `bind` passes the modified world from one action into the next action.

Because the entire universe doesn't fit in a computer's memory, the world being passed around is just a representation. So long as world tokens are not re-used, the representation is safe. This means that world tokens do not need to contain any data at all:

```
#print IO.RealWorld
```

```
def IO.RealWorld : Type :=
Unit
```

Additional Conveniences

Shared Argument Types

When defining a function that takes multiple arguments that have the same type, both can be written before the same colon. For example,

```
def equal? [BEq α] (x : α) (y : α) : Option α :=
  if x == y then
    some x
  else
    none
```

can be written

```
def equal? [BEq α] (x y : α) : Option α :=
  if x == y then
    some x
  else
    none
```

This is especially useful when the type signature is large.

Leading Dot Notation

The constructors of an inductive type are in a namespace. This allows multiple related inductive types to use the same constructor names, but it can lead to programs becoming verbose. In contexts where the inductive type in question is known, the namespace can be omitted by preceding the constructor's name with a dot, and Lean uses the expected type to resolve the constructor names. For example, a function that mirrors a binary tree can be written:

```
def BinTree.mirror : BinTree α → BinTree α
  | BinTree.leaf => BinTree.leaf
  | BinTree.branch l x r => BinTree.branch (mirror r) x (mirror l)
```

Omitting the namespaces makes it significantly shorter, at the cost of making the program harder to read in contexts like code review tools that don't include the Lean compiler:

```
def BinTree.mirror : BinTree α → BinTree α
  | .leaf => .leaf
  | .branch l x r => .branch (mirror r) x (mirror l)
```

Using the expected type of an expression to disambiguate a namespace is also applicable to names other than constructors. If `BinTree.empty` is defined as an alternative way of creating `BinTree` s, then it can also be used with dot notation:

```
def BinTree.empty : BinTree α := .leaf
```

```
#check (.empty : BinTree Nat)
```

```
BinTree.empty : BinTree Nat
```

Or-Patterns

In contexts that allow multiple patterns, such as `match` -expressions, multiple patterns may share their result expressions. The datatype `Weekday` that represents days of the week:

```
inductive Weekday where
  | monday
  | tuesday
  | wednesday
  | thursday
  | friday
  | saturday
  | sunday
  deriving Repr
```

Pattern matching can be used to check whether a day is a weekend:

```
def Weekday.isWeekend (day : Weekday) : Bool :=
  match day with
  | Weekday.saturday => true
  | Weekday.sunday => true
  | _ => false
```

This can already be simplified by using constructor dot notation:

```
def Weekday.isWeekend (day : Weekday) : Bool :=
  match day with
  | .saturday => true
  | .sunday => true
  | _ => false
```

Because both weekend patterns have the same result expression (`true`), they can be condensed into one:

```
def Weekday.isWeekend (day : Weekday) : Bool :=
  match day with
  | .saturday | .sunday => true
  | _ => false
```

This can be further simplified into a version in which the argument is not named:

```
def Weekday.isWeekend : Weekday → Bool
  | .saturday | .sunday => true
  | _ => false
```

Behind the scenes, the result expression is simply duplicated across each pattern. This means that patterns can bind variables, as in this example that removes the `inl` and `inr` constructors from a sum type in which both contain the same type of value:

```
def condense : α ⊕ α → α
  | .inl x | .inr x => x
```

Because the result expression is duplicated, the variables bound by the patterns are not required to have the same types. Overloaded functions that work for multiple types may be used to write a single result expression that works for patterns that bind variables of different types:

```
def stringy : Nat ⊕ Weekday → String
  | .inl x | .inr x => s!"It is {repr x}"
```

In practice, only variables shared in all patterns can be referred to in the result expression, because the result must make sense for each pattern. In `getTheNat`, only `n` can be accessed, and attempts to use either `x` or `y` lead to errors.

```
def getTheNat : (Nat × α) ⊕ (Nat × β) → Nat
  | .inl (n, x) | .inr (n, y) => n
```

Attempting to access `x` in a similar definition causes an error because there is no `x` available in the second pattern:

```
def getTheAlpha : (Nat × α) ⊕ (Nat × α) → α
  | .inl (n, x) | .inr (n, y) => x
```

```
unknown identifier 'x'
```

The fact that the result expression is essentially copy-pasted to each branch of the pattern match can lead to some surprising behavior. For example, the following definitions are acceptable because the `inr` version of the result expression refers to the global definition of `str`:

```
def str := "Some string"

def getTheString : (Nat × String) ⊕ (Nat × β) → String
  | .inl (n, str) | .inr (n, y) => str
```

Calling this function on both constructors reveals the confusing behavior. In the first case, a type annotation is needed to tell Lean which type `β` should be:

```
#eval getTheString (.inl (20, "twenty") : (Nat × String) ⊕ (Nat × String))
```

```
"twenty"
```

In the second case, the global definition is used:

```
#eval getTheString (.inr (20, "twenty"))
```

```
"Some string"
```

Using or-patterns can vastly simplify some definitions and increase their clarity, as in `Weekday.isWeekend`. Because there is a potential for confusing behavior, it's a good idea to be careful when using them, especially when variables of multiple types or disjoint sets of variables are involved.

Summary

Encoding Side Effects

Lean is a pure functional language. This means that it does not include side effects such as mutable variables, logging, or exceptions. However, most side effects can be *encoded* using a combination of functions and inductive types or structures. For example, mutable state can be encoded as a function from an initial state to a pair of a final state and a result, and exceptions can be encoded as an inductive type with constructors for successful termination and errors.

Each set of encoded effects is a type. As a result, if a program uses these encoded effects, then this is apparent in its type. Functional programming does not mean that programs can't use effects, it simply requires that they be *honest* about which effects they use. A Lean type signature describes not only the types of arguments that a function expects and the type of result that it returns, but also which effects it may use.

The Monad Type Class

It's possible to write purely functional programs in languages that allow effects anywhere. For example, `2 + 3` is a valid Python program that has no effects at all. Similarly, combining programs that have effects requires a way to state the order in which the effects must occur. It matters whether an exception is thrown before or after modifying a variable, after all.

The type class `Monad` captures these two important properties. It has two methods: `pure` represents programs that have no effects, and `bind` sequences effectful programs. The contract for `Monad` instances ensures that `bind` and `pure` actually capture pure computation and sequencing.

do-Notation for Monads

Rather than being limited to `IO`, `do`-notation works for any monad. It allows programs that use monads to be written in a style that is reminiscent of statement-oriented languages, with statements sequenced after one another. Additionally, `do`-notation enables a number of additional convenient shorthands, such as nested actions. A program written with `do` is translated to applications of `>>=` behind the scenes.

Custom Monads

Different languages provide different sets of side effects. While most languages feature mutable variables and file I/O, not all have features like exceptions. Other languages offer effects that are rare or unique, like Icon's search-based program execution, Scheme and Ruby's continuations, and Common Lisp's resumable exceptions. An advantage to encoding effects with monads is that programs are not limited to the set of effects that are provided by the language. Because Lean is designed to make programming with any monad convenient, programmers are free to choose exactly the set of side effects that make sense for any given application.

The `IO` Monad

Programs that can affect the real world are written as `IO` actions in Lean. `IO` is one monad among many. The `IO` monad encodes state and exceptions, with the state being used to keep track of the state of the world and the exceptions modeling failure and recovery.

Functors, Applicative Functors, and Monads

`Functor` and `Monad` both describe operations for types that are still waiting for a type argument. One way to understand them is that `Functor` describes containers in which the contained data can be transformed, and `Monad` describes an encoding of programs with side effects. This understanding is incomplete, however. After all, `Option` has instances for both `Functor` and `Monad`, and simultaneously represents an optional value *and* a computation that might fail to return a value.

From the perspective of data structures, `Option` is a bit like a nullable type or like a list that can contain at most one entry. From the perspective of control structures, `Option` represents a computation that might terminate early without a result. Typically, programs that use the `Functor` instance are easiest to think of as using `Option` as a data structure, while programs that use the `Monad` instance are easiest to think of as using `Option` to allow early failure, but learning to use both of these perspectives fluently is an important part of becoming proficient at functional programming.

There is a deeper relationship between functors and monads. It turns out that *every monad is a functor*. Another way to say this is that the monad abstraction is more powerful than the functor abstraction, because not every functor is a monad. Furthermore, there is an additional intermediate abstraction, called *applicative functors*, that has enough power to write many interesting programs and yet permits libraries that cannot use the `Monad` interface. The type class `Applicative` provides the overloadable operations of applicative functors. Every monad is an applicative functor, and every applicative functor is a functor, but the converses do not hold.

Structures and Inheritance

In order to understand the full definitions of `Functor`, `Applicative`, and `Monad`, another Lean feature is necessary: structure inheritance. Structure inheritance allows one structure type to provide the interface of another, along with additional fields. This can be useful when modeling concepts that have a clear taxonomic relationship. For example, take a model of mythical creatures. Some of them are large, and some are small:

```
structure MythicalCreature where
  large : Bool
deriving Repr
```

Behind the scenes, defining the `MythicalCreature` structure creates an inductive type with a single constructor called `mk`:

```
#check MythicalCreature.mk
```

```
MythicalCreature.mk (large : Bool) : MythicalCreature
```

Similarly, a function `MythicalCreature.large` is created that actually extracts the field from the constructor:

```
#check MythicalCreature.large
```

```
MythicalCreature.large (self : MythicalCreature) : Bool
```

In most old stories, each monster can be defeated in some way. A description of a monster should include this information, along with whether it is large:

```
structure Monster extends MythicalCreature where
  vulnerability : String
deriving Repr
```

The `extends MythicalCreature` in the heading states that every monster is also mythical. To define a `Monster`, both the fields from `MythicalCreature` and the fields from `Monster` should be provided. A troll is a large monster that is vulnerable to sunlight:

```
def troll : Monster where
  large := true
  vulnerability := "sunlight"
```

Behind the scenes, inheritance is implemented using composition. The constructor

`Monster.mk` takes a `MythicalCreature` as its argument:

```
#check Monster.mk
```

```
Monster.mk (toMythicalCreature : MythicalCreature) (vulnerability : String) :
Monster
```

In addition to defining functions to extract the value of each new field, a function `Monster.toMythicalCreature` is defined with type `Monster → MythicalCreature`. This can be used to extract the underlying creature.

Moving up the inheritance hierarchy in Lean is not the same thing as upcasting in object-oriented languages. An upcast operator causes a value from a derived class to be treated as an instance of the parent class, but the value retains its identity and structure. In Lean, however, moving up the inheritance hierarchy actually erases the underlying information. To see this in action, consider the result of evaluating `troll.toMythicalCreature`:

```
#eval troll.toMythicalCreature
```

```
{ large := true }
```

Only the fields of `MythicalCreature` remain.

Just like the `where` syntax, curly-brace notation with field names also works with structure inheritance:

```
def troll : Monster := {large := true, vulnerability := "sunlight"}
```

However, the anonymous angle-bracket notation that delegates to the underlying constructor reveals the internal details:

```
def troll : Monster := ⟨true, "sunlight"⟩
```

```
application type mismatch
  Monster.mk true
argument
  true
has type
  Bool : Type
but is expected to have type
  MythicalCreature : Type
```

An extra set of angle brackets is required, which invokes `MythicalCreature.mk` on `true`:

```
def troll : Monster := ⟨⟨true⟩, "sunlight"⟩
```

Lean's dot notation is capable of taking inheritance into account. In other words, the existing `MythicalCreature.large` can be used with a `Monster`, and Lean automatically inserts the call to `Monster.toMythicalCreature` before the call to `MythicalCreature.large`. However, this only occurs when using dot notation, and applying the field lookup function using normal function call syntax results in a type error:

```
#eval MythicalCreature.large troll
```

```
application type mismatch
  MythicalCreature.large troll
argument
  troll
has type
  Monster : Type
but is expected to have type
  MythicalCreature : Type
```

Dot notation can also take inheritance into account for user-defined functions. A small creature is one that is not large:

```
def MythicalCreature.small (c : MythicalCreature) : Bool := !c.large
```

Evaluating `troll.small` yields `false`, while attempting to evaluate `MythicalCreature.small troll` results in:

```
application type mismatch
  MythicalCreature.small troll
argument
  troll
has type
  Monster : Type
but is expected to have type
  MythicalCreature : Type
```

Multiple Inheritance

A helper is a mythical creature that can provide assistance when given the correct payment:

```
structure Helper extends MythicalCreature where
  assistance : String
  payment : String
deriving Repr
```

For example, a *nisse* is a kind of small elf that's known to help around the house when provided with tasty porridge:

```
def nisse : Helper where
  large := false
  assistance := "household tasks"
  payment := "porridge"
```

If domesticated, trolls make excellent helpers. They are strong enough to plow a whole field in a single night, though they require model goats to keep them satisfied with their lot in life. A monstrous assistant is a monster that is also a helper:

```
structure MonstrousAssistant extends Monster, Helper where
deriving Repr
```

A value of this structure type must fill in all of the fields from both parent structures:

```
def domesticatedTroll : MonstrousAssistant where
  large := true
  assistance := "heavy labor"
  payment := "toy goats"
  vulnerability := "sunlight"
```

Both of the parent structure types extend `MythicalCreature`. If multiple inheritance were implemented naïvely, then this could lead to a "diamond problem", where it would be unclear which path to `large` should be taken from a given `MonstrousAssistant`. Should it take `large` from the contained `Monster` or from the contained `Helper`? In Lean, the answer is that the first specified path to the grandparent structure is taken, and the additional parent structures' fields are copied rather than having the new structure include both parents directly.

This can be seen by examining the signature of the constructor for `MonstrousAssistant`:

```
#check MonstrousAssistant.mk
```

```
MonstrousAssistant.mk (toMonster : Monster) (assistance payment : String) :
MonstrousAssistant
```

It takes a `Monster` as an argument, along with the two fields that `Helper` introduces on top of `MythicalCreature`. Similarly, while `MonstrousAssistant.toMonster` merely extracts the `Monster` from the constructor, `MonstrousAssistant.toHelper` has no `Helper` to extract. The `#print` command exposes its implementation:

```
#print MonstrousAssistant.toHelper
```

```
@[reducible] def MonstrousAssistant.toHelper : MonstrousAssistant → Helper :=
fun self => { toMythicalCreature := self.toMythicalCreature, assistance :=
self.assistance, payment := self.payment }
```

This function constructs a `Helper` from the fields of `MonstrousAssistant`. The `@[reducible]` attribute has the same effect as writing `abbrev`.

Default Declarations

When one structure inherits from another, default field definitions can be used to instantiate the parent structure's fields based on the child structure's fields. If more size specificity is required than whether a creature is large or not, a dedicated datatype describing sizes can be used together with inheritance, yielding a structure in which the `large` field is computed from the contents of the `size` field:

```
inductive Size where
  | small
  | medium
  | large
deriving BEq

structure SizedCreature extends MythicalCreature where
  size : Size
  large := size == Size.large
```

This default definition is only a default definition, however. Unlike property inheritance in a language like C# or Scala, the definitions in the child structure are only used when no specific value for `large` is provided, and nonsensical results can occur:

```
def nonsenseCreature : SizedCreature where
  large := false
  size := .large
```

If the child structure should not deviate from the parent structure, there are a few options:

1. Documenting the relationship, as is done for `BEq` and `Hashable`
2. Defining a proposition that the fields are related appropriately, and designing the API to require evidence that the proposition is true where it matters
3. Not using inheritance at all

The second option could look like this:

```
abbrev SizesMatch (sc : SizedCreature) : Prop :=
  sc.large = (sc.size == Size.large)
```

Note that a single equality sign is used to indicate the equality *proposition*, while a double equality sign is used to indicate a function that checks equality and returns a `Bool`. `SizesMatch` is defined as an `abbrev` because it should automatically be unfolded in proofs, so that `simp` can see the equality that should be proven.

A *huldre* is a medium-sized mythical creature—in fact, they are the same size as humans. The two sized fields on `huldre` match one another:

```
def huldre : SizedCreature where
  size := .medium

example : SizesMatch huldre := by
  decide
```

Type Class Inheritance

Behind the scenes, type classes are structures. Defining a new type class defines a new structure, and defining an instance creates a value of that structure type. They are then added to internal tables in Lean that allow it to find the instances upon request. A consequence of this is that type classes may inherit from other type classes.

Because it uses precisely the same language features, type class inheritance supports all the features of structure inheritance, including multiple inheritance, default implementations of parent types' methods, and automatic collapsing of diamonds. This is useful in many of the same situations that multiple interface inheritance is useful in languages like Java, C# and Kotlin. By carefully designing type class inheritance hierarchies, programmers can get the best of both worlds: a fine-grained collection of independently-implementable abstractions, and automatic construction of these specific abstractions from larger, more general abstractions.

Applicative Functors

An *applicative functor* is a functor that has two additional operations available: `pure` and `seq`. `pure` is the same operator used in `Monad`, because `Monad` in fact inherits from `Applicative`. `seq` is much like `map`: it allows a function to be used in order to transform the contents of a datatype. However, with `seq`, the function is itself contained in the datatype: `f (α → β) → (Unit → f α) → f β`. Having the function under the type `f` allows the `Applicative` instance to control how the function is applied, while `Functor.map` unconditionally applies a function. The second argument has a type that begins with `Unit →` to allow the definition of `seq` to short-circuit in cases where the function will never be applied.

The value of this short-circuiting behavior can be seen in the instance of `Applicative Option`:

```
instance : Applicative Option where
  pure x := .some x
  seq f x :=
    match f with
    | none => none
    | some g => g <$> x ()
```

In this case, if there is no function for `seq` to apply, then there is no need to compute its argument, so `x` is never called. The same consideration informs the instance of `Applicative` for `Except`:

```
instance : Applicative (Except ε) where
  pure x := .ok x
  seq f x :=
    match f with
    | .error e => .error e
    | .ok g => g <$> x ()
```

This short-circuiting behavior depends only on the `Option` or `Except` structures that *surround* the function, rather than on the function itself.

Monads can be seen as a way of capturing the notion of sequentially executing statements into a pure functional language. The result of one statement can affect which further statements run. This can be seen in the type of `bind`: `m α → (α → m β) → m β`. The first statement's resulting value is an input into a function that computes the next statement to execute. Successive uses of `bind` are like a sequence of statements in an imperative programming language, and `bind` is powerful enough to implement control structures like conditionals and loops.

Following this analogy, `Applicative` captures function application in a language that has side effects. The arguments to a function in languages like Kotlin or C# are evaluated from left to right. Side effects performed by earlier arguments occur before those performed by later arguments. A function is not powerful enough to implement custom short-circuiting operators that depend on the specific *value* of an argument, however.

Typically, `seq` is not invoked directly. Instead, the operator `<*>` is used. This operator wraps its second argument in `fun () => ...`, simplifying the call site. In other words, `E1 <*> E2` is syntactic sugar for `Seq.seq E1 (fun () => E2)`.

The key feature that allows `seq` to be used with multiple arguments is that a multiple-argument Lean function is really a single-argument function that returns another function that's waiting for the rest of the arguments. In other words, if the first argument to `seq` is awaiting multiple arguments, then the result of the `seq` will be awaiting the rest. For example, `some Plus.plus` can have the type `Option (Nat → Nat → Nat)`. Providing one argument, `some Plus.plus <*> some 4`, results in the type `Option (Nat → Nat)`. This can itself be used with `seq`, so `some Plus.plus <*> some 4 <*> some 7` has the type `Option Nat`.

Not every functor is applicative. `Pair` is like the built-in product type `Prod`:

```
structure Pair (α β : Type) : Type where
  first : α
  second : β
```

Like `Except`, `Pair` has type `Type → Type → Type`. This means that `Pair α` has type `Type → Type`, and a `Functor` instance is possible:

```
instance : Functor (Pair α) where
  map f x := ⟨x.first, f x.second⟩
```

This instance obeys the `Functor` contract.

The two properties to check are that `id <$> Pair.mk x y = Pair.mk x y` and that `f <$> g <$> Pair.mk x y = (f ∘ g) <$> Pair.mk x y`. The first property can be checked by just stepping through the evaluation of the left side, and noticing that it evaluates to the right side:

```
id <$> Pair.mk x y
===>
Pair.mk x (id y)
===>
Pair.mk x y
```

The second can be checked by stepping through both sides, and noting that they yield the same result:

```
f <$> g <$> Pair.mk x y
===>
f <$> Pair.mk x (g y)
===>
Pair.mk x (f (g y))

(f ∘ g) <$> Pair.mk x y
===>
Pair.mk x ((f ∘ g) y)
===>
Pair.mk x (f (g y))
```

Attempting to define an `Applicative` instance, however, does not work so well. It will require a definition of `pure`:

```
def Pair.pure (x : β) : Pair α β := _
```

```
don't know how to synthesize placeholder
context:
β α : Type
x : β
⊢ Pair α β
```

There is a value with type `β` in scope (namely `x`), and the error message from the underscore suggests that the next step is to use the constructor `Pair.mk`:

```
def Pair.pure (x : β) : Pair α β := Pair.mk _ x
```

```
don't know how to synthesize placeholder for argument 'first'
context:
β α : Type
x : β
⊢ α
```

Unfortunately, there is no `α` available. Because `pure` would need to work for *all possible types* `α` to define an instance of `Applicative (Pair α)`, this is impossible. After all, a caller could choose `α` to be `Empty`, which has no values at all.

A Non-Monadic Applicative

When validating user input to a form, it's generally considered to be best to provide many errors at once, rather than one error at a time. This allows the user to have an overview of what is needed to please the computer, rather than feeling badgered as they correct the errors field by field.

Ideally, validating user input will be visible in the type of the function that's doing the validating. It should return a datatype that is specific—checking that a text box contains a number should return an actual numeric type, for instance. A validation routine could throw an exception when the input does not pass validation. Exceptions have a major drawback, however: they terminate the program at the first error, making it impossible to accumulate a list of errors.

On the other hand, the common design pattern of accumulating a list of errors and then failing when it is non-empty is also problematic. A long nested sequences of `if` statements that validate each sub-section of the input data is hard to maintain, and it's easy to lose track of an error message or two. Ideally, validation can be performed using an API that enables a new value to be returned yet automatically tracks and accumulates error messages.

An applicative functor called `Validate` provides one way to implement this style of API. Like the `Except` monad, `Validate` allows a new value to be constructed that characterizes the validated data accurately. Unlike `Except`, it allows multiple errors to be accumulated, without a risk of forgetting to check whether the list is empty.

User Input

As an example of user input, take the following structure:

```
structure RawInput where
  name : String
  birthYear : String
```

The business logic to be implemented is the following:

1. The name may not be empty
2. The birth year must be numeric and non-negative
3. The birth year must be greater than 1900, and less than or equal to the year in which the form is validated

Representing these as a datatype will require a new feature, called *subtypes*. With this tool in hand, a validation framework can be written that uses an applicative functor to track errors, and these rules can be implemented in the framework.

Subtypes

Representing these conditions is easiest with one additional Lean type, called `Subtype`:

```
structure Subtype {α : Type} (p : α → Prop) where
  val : α
  property : p val
```

This structure has two type parameters: an implicit parameter that is the type of data `α`, and an explicit parameter `p` that is a predicate over `α`. A *predicate* is a logical statement with a variable in it that can be replaced with a value to yield an actual statement, like the parameter to `GetElem` that describes what it means for an index to be in bounds for a lookup. In the case of `Subtype`, the predicate slices out some subset of the values of `α` for which the predicate holds. The structure's two fields are, respectively, a value from `α` and evidence that the value satisfies the predicate `p`. Lean has special syntax for `Subtype`. If `p` has type `α → Prop`, then the type `Subtype p` can also be written `{x : α // p x}`, or even `{x // p x}` when the type can be inferred automatically.

Representing positive numbers as inductive types is clear and easy to program with. However, it has a key disadvantage. While `Nat` and `Int` have the structure of ordinary inductive types from the perspective of Lean programs, the compiler treats them specially and uses fast arbitrary-precision number libraries to implement them. This is not the case for additional user-defined types. However, a subtype of `Nat` that restricts it to non-zero numbers allows the new type to use the efficient representation while still ruling out zero at compile time:

```
def FastPos : Type := {x : Nat // x > 0}
```

The smallest fast positive number is still one. Now, instead of being a constructor of an inductive type, it's an instance of a structure that's constructed with angle brackets. The first argument is the underlying `Nat`, and the second argument is the evidence that said `Nat` is greater than zero:

```
def one : FastPos := ⟨1, by decide⟩
```

The proposition `1 > 0` is decidable, so the `decide` tactic produces the necessary evidence. The `OfNat` instance is very much like that for `Pos`, except it uses a short tactic proof to provide evidence that `n + 1 > 0`:

```
instance : OfNat FastPos (n + 1) where
  ofNat := ⟨n + 1, by simp⟩
```

Here, `simp` is needed because `decide` requires concrete values, but the proposition in question is `n + 1 > 0`.

Subtypes are a two-edged sword. They allow efficient representation of validation rules, but they transfer the burden of maintaining these rules to the users of the library, who have to *prove* that they are not violating important invariants. Generally, it's a good idea to use them internally to a library, providing an API to users that automatically ensures that all invariants are satisfied, with any necessary proofs being internal to the library.

Checking whether a value of type `α` is in the subtype `{x : α // p x}` usually requires that the proposition `p x` be decidable. The section on equality and ordering classes describes how decidable propositions can be used with `if`. When `if` is used with a decidable proposition, a name can be provided. In the `then` branch, the name is bound to evidence that the proposition is true, and in the `else` branch, it is bound to evidence that the proposition is false. This comes in handy when checking whether a given `Nat` is positive:

```
def Nat.asFastPos? (n : Nat) : Option FastPos :=
  if h : n > 0 then
    some ⟨n, h⟩
  else none
```

In the `then` branch, `h` is bound to evidence that `n > 0`, and this evidence can be used as the second argument to `Subtype`'s constructor.

Validated Input

The validated user input is a structure that expresses the business logic using multiple techniques:

- The structure type itself encodes the year in which it was checked for validity, so that `CheckedInput 2019` is not the same type as `CheckedInput 2020`
- The birth year is represented as a `Nat` rather than a `String`
- Subtypes are used to constrain the allowed values in the name and birth year fields

```
structure CheckedInput (thisYear : Nat) : Type where
  name : {n : String // n ≠ ""}
  birthYear : {y : Nat // y > 1900 ∧ y ≤ thisYear}
```

An input validator should take the current year and a `RawInput` as arguments, returning either a checked input or at least one validation failure. This is represented by the `Validate` type:

```
inductive Validate (ε α : Type) : Type where
  | ok : α → Validate ε α
  | errors : NonEmptyList ε → Validate ε α
```

It looks very much like `Except`. The only difference is that the `error` constructor may contain more than one failure.

`Validate` is a functor. Mapping a function over it transforms any successful value that might be present, just as in the `Functor` instance for `Except`:

```
instance : Functor (Validate ε) where
  map f
   | .ok x => .ok (f x)
   | .errors errs => .errors errs
```

The `Applicative` instance for `Validate` has an important difference from the instance for `Except`: while the instance for `Except` terminates at the first error encountered, the instance for `Validate` is careful to accumulate all errors from *both* the function and the argument branches:

```
instance : Applicative (Validate ε) where
  pure := .ok
  seq f x :=
    match f with
    | .ok g => g <$> (x ())
    | .errors errs =>
      match x () with
      | .ok _ => .errors errs
      | .errors errs' => .errors (errs ++ errs')
```

Using `.errors` together with the constructor for `NonEmptyList` is a bit verbose. Helpers like `reportError` make code more readable. In this application, error reports will consist of field names paired with messages:

```
def Field := String

def reportError (f : Field) (msg : String) : Validate (Field × String) α :=
  .errors { head := (f, msg), tail := [] }
```

The `Applicative` instance for `Validate` allows the checking procedures for each field to be written independently and then composed. Checking a name consists of ensuring that a string is non-empty, then returning evidence of this fact in the form of a `Subtype`. This uses the evidence-binding version of `if`:

```
def checkName (name : String) : Validate (Field × String) {n : String // n ≠ ""}
:=
  if h : name = "" then
    reportError "name" "Required"
  else pure ⟨name, h⟩
```

In the `then` branch, `h` is bound to evidence that `name = ""`, while it is bound to evidence that `¬name = ""` in the `else` branch.

It's certainly the case that some validation errors make other checks impossible. For example, it makes no sense to check whether the birth year field is greater than 1900 if a confused user wrote the word `"syzygy"` instead of a number. Checking the allowed range of the number is only meaningful after ensuring that the field in fact contains a number. This can be expressed using the function `andThen`:

```
def Validate.andThen (val : Validate ε α) (next : α → Validate ε β) : Validate ε
β :=
  match val with
  | .errors errs => .errors errs
  | .ok x => next x
```

While this function's type signature makes it suitable to be used as `bind` in a `Monad` instance, there are good reasons not to do so. They are described in the section that describes the `Applicative` contract.

To check that the birth year is a number, a built-in function called `String.toNat? : String → Option Nat` is useful. It's most user-friendly to eliminate leading and trailing whitespace first using `String.trim`:

```
def checkYearIsNat (year : String) : Validate (Field × String) Nat :=
  match year.trim.toNat? with
  | none => reportError "birth year" "Must be digits"
  | some n => pure n
```

To check that the provided year is in the expected range, nested uses of the evidence-providing form of `if` are in order:

```
def checkBirthYear (thisYear year : Nat) : Validate (Field × String) {y : Nat //
y > 1900 ∧ y ≤ thisYear} :=
  if h : year > 1900 then
    if h' : year ≤ thisYear then
      pure ⟨year, by simp [*]⟩
    else reportError "birth year" s!"Must be no later than {thisYear}"
  else reportError "birth year" "Must be after 1900"
```

Finally, these three components can be combined using `seq`:

```
def checkInput (year : Nat) (input : RawInput) : Validate (Field × String)
(CheckedInput year) :=
  pure CheckedInput.mk <*>
    checkName input.name <*>
    (checkYearIsNat input.birthYear).andThen fun birthYearAsNat =>
      checkBirthYear year birthYearAsNat
```

Testing `checkInput` shows that it can indeed return multiple pieces of feedback:

```
#eval checkInput 2023 {name := "David", birthYear := "1984"}
```

```
Validate.ok { name := "David", birthYear := 1984 }
```

```
#eval checkInput 2023 {name := "", birthYear := "2045"}
```

```
Validate.errors { head := ("name", "Required"), tail := [("birth year", "Must be
no later than 2023")] }
```

```
#eval checkInput 2023 {name := "David", birthYear := "syzygy"}
```

```
Validate.errors { head := ("birth year", "Must be digits"), tail := [] }
```

Form validation with `checkInput` illustrates a key advantage of `Applicative` over `Monad`. Because `>>=` provides enough power to modify the rest of the program's execution based on the value from the first step, it *must* receive a value from the first step to pass on. If no value is received (e.g. because an error has occurred), then `>>=` cannot execute the rest of the program. `Validate` demonstrates why it can be useful to run the rest of the program anyway: in cases where the earlier data isn't needed, running the rest of the program can yield useful information (in this case, more validation errors). `Applicative`'s `<*>` may run both of its arguments before recombining the results. Similarly, `>>=` forces sequential execution. Each step must complete before the next may run. This is generally useful, but it makes it impossible to have parallel execution of different threads that naturally emerges from the program's actual data dependencies. A more powerful abstraction like `Monad` increases the flexibility that's available to the API consumer, but it decreases the flexibility that is available to the API implementor.

The Applicative Contract

Just like `Functor`, `Monad`, and types that implement `BEq` and `Hashable`, `Applicative` has a set of rules that all instances should adhere to.

There are four rules that an applicative functor should follow:

1. It should respect identity, so `pure id <*> v = v`
2. It should respect function composition, so `pure (· ∘ ·) <*> u <*> v <*> w = u <*> (v <*> w)`
3. Sequencing pure operations should be a no-op, so `pure f <*> pure x = pure (f x)`
4. The ordering of pure operations doesn't matter, so `u <*> pure x = pure (fun f => f x) <*> u`

To check these for the `Applicative Option` instance, start by expanding `pure` into `some`.

The first rule states that `some id <*> v = v`. The definition of `seq` for `Option` states that this is the same as `id <$> v = v`, which is one of the `Functor` rules that have already been checked.

The second rule states that `some (· ∘ ·) <*> u <*> v <*> w = u <*> (v <*> w)`. If any of `u`, `v`, or `w` is `none`, then both sides are `none`, so the property holds. Assuming that `u` is `some f`, that `v` is `some g`, and that `w` is `some x`, then this is equivalent to saying that `some (· ∘ ·) <*> some f <*> some g <*> some x = some f <*> (some g <*> some x)`. Evaluating the two sides yields the same result:

```
some (· ∘ ·) <*> some f <*> some g <*> some x
===>
some (f ∘ ·) <*> some g <*> some x
===>
some (f ∘ g) <*> some x
===>
some ((f ∘ g) x)
===>
some (f (g x))

some f <*> (some g <*> some x)
===>
some f <*> (some (g x))
===>
some (f (g x))
```

The third rule follows directly from the definition of `seq`:

```
some f <*> some x
===>
f <$> some x
===>
some (f x)
```

In the fourth case, assume that `u` is `some f`, because if it's `none`, both sides of the equation are `none`. `some f <*> some x` evaluates directly to `some (f x)`, as does `some (fun g => g x) <*> some f`.

All Applicatives are Functors

The two operators for `Applicative` are enough to define `map`:

```
def map [Applicative f] (g : α → β) (x : f α) : f β :=
  pure g <*> x
```

This can only be used to implement `Functor` if the contract for `Applicative` guarantees the contract for `Functor`, however. The first rule of `Functor` is that `id <$> x = x`, which follows directly from the first rule for `Applicative`. The second rule of `Functor` is that `map (f ∘ g) x = map f (map g x)`. Unfolding the definition of `map` here results in `pure (f ∘ g) <*> x = pure f <*> (pure g <*> x)`. Using the rule that sequencing pure operations is a no-op, the left side can be rewritten to `pure (· ∘ ·) <*> pure f <*> pure g <*> x`. This is an instance of the rule that states that applicative functors respect function composition.

This justifies a definition of `Applicative` that extends `Functor`, with a default definition of `map` given in terms of `pure` and `seq`:

```
class Applicative (f : Type → Type) extends Functor f where
  pure : α → f α
  seq : f (α → β) → (Unit → f α) → f β
  map g x := seq (pure g) (fun () => x)
```

All Monads are Applicative Functors

An instance of `Monad` already requires an implementation of `pure`. Together with `bind`, this is enough to define `seq`:

```
def seq [Monad m] (f : m (α → β)) (x : Unit → m α) : m β := do
  let g ← f
  let y ← x ()
  pure (g y)
```

Once again, checking that the `Monad` contract implies the `Applicative` contract will allow this to be used as a default definition for `seq` if `Monad` extends `Applicative`.

The rest of this section consists of an argument that this implementation of `seq` based on `bind` in fact satisfies the `Applicative` contract. One of the beautiful things about functional programming is that this kind of argument can be worked out on a piece of paper with a pencil, using the kinds of evaluation rules from the initial section on evaluating expressions. Thinking about the meanings of the operations while reading these arguments can sometimes help with understanding.

Replacing `do`-notation with explicit uses of `>>=` makes it easier to apply the `Monad` rules:

```
def seq [Monad m] (f : m (α → β)) (x : Unit → m α) : m β := do
  f >>= fun g =>
  x () >>= fun y =>
  pure (g y)
```

To check that this definition respects identity, check that `seq (pure id) (fun () => v) = v`. The left hand side is equivalent to `pure id >>= fun g => (fun () => v) () >>= fun y => pure (g y)`. The unit function in the middle can be eliminated immediately, yielding `pure id >>= fun g => v >>= fun y => pure (g y)`. Using the fact that `pure` is a left identity of `>>=`, this is the same as `v >>= fun y => pure (id y)`, which is `v >>= fun y => pure y`. Because `fun x => f x` is the same as `f`, this is the same as `v >>= pure`, and the fact that `pure` is a right identity of `>>=` can be used to get `v`.

This kind of informal reasoning can be made easier to read with a bit of reformatting. In the following chart, read "EXPR1 ={ REASON }= EXPR2" as "EXPR1 is the same as EXPR2 because REASON":

`pure id >>= fun g => v >>= fun y => pure (g y)`

={ `pure` *is a left identity of* `>>=` }=

`v >>= fun y => pure (id y)`

={ *Reduce the call to* `id` }=

`v >>= fun y => pure y`

={ `fun x => f x` *is the same as* `f` }=

`v >>= pure`

={ `pure` *is a right identity of* `>>=` }=

```
v
```

To check that it respects function composition, check that `pure (· ∘ ·) <*> u <*> v <*> w = u <*> (v <*> w)`. The first step is to replace `<*>` with this definition of `seq`. After that, a (somewhat long) series of steps that use the identity and associativity rules from the `Monad` contract is enough to get from one to the other:

```
seq (seq (seq (pure (· ∘ ·)) (fun _ => u))
      (fun _ => v))
  (fun _ => w)
```

={ *Definition of* `seq` }=

```
((pure (· ∘ ·) >>= fun f =>
   u >>= fun x =>
   pure (f x)) >>= fun g =>
  v >>= fun y =>
  pure (g y)) >>= fun h =>
 w >>= fun z =>
 pure (h z)
```

={ `pure` *is a left identity of* `>>=` }=

```
((u >>= fun x =>
   pure (x ∘ ·)) >>= fun g =>
   v >>= fun y =>
  pure (g y)) >>= fun h =>
 w >>= fun z =>
 pure (h z)
```

={ *Insertion of parentheses for clarity* }=

```
((u >>= fun x =>
   pure (x ∘ ·)) >>= (fun g =>
   v >>= fun y =>
  pure (g y))) >>= fun h =>
 w >>= fun z =>
 pure (h z)
```

={ *Associativity of* `>>=` }=

```
(u >>= fun x =>
  pure (x ∘ ·) >>= fun g =>
 v  >>= fun y => pure (g y)) >>= fun h =>
 w >>= fun z =>
 pure (h z)
```

={ `pure` *is a left identity of* `>>=` }=

```
(u >>= fun x =>
  v >>= fun y =>
  pure (x ∘ y)) >>= fun h =>
 w >>= fun z =>
 pure (h z)
```

={ *Associativity of* `>>=` }=

```
u >>= fun x =>
v >>= fun y =>
pure (x ∘ y) >>= fun h =>
w >>= fun z =>
pure (h z)
```

={ `pure` *is a left identity of* `>>=` }=

```
u >>= fun x =>
v >>= fun y =>
w >>= fun z =>
pure ((x ∘ y) z)
```

={ *Definition of function composition* }=

```
u >>= fun x =>
v >>= fun y =>
w >>= fun z =>
pure (x (y z))
```

={ *Time to start moving backwards!* `pure` *is a left identity of* `>>=` }=

```
u >>= fun x =>
v >>= fun y =>
w >>= fun z =>
pure (y z) >>= fun q =>
pure (x q)
```

={ *Associativity of* `>>=` }=

```
u >>= fun x =>
v >>= fun y =>
 (w >>= fun p =>
  pure (y p)) >>= fun q =>
 pure (x q)
```

={ *Associativity of* `>>=` }=

```
u >>= fun x =>
 (v >>= fun y =>
  w >>= fun q =>
  pure (y q)) >>= fun z =>
 pure (x z)
```

={ *This includes the definition of* `seq` }=

```
u >>= fun x =>
seq v (fun () => w) >>= fun q =>
pure (x q)
```

={ *This also includes the definition of* `seq` }=

```
seq u (fun () => seq v (fun () => w))
```

To check that sequencing pure operations is a no-op:

```
seq (pure f) (fun () => pure x)
```

={ *Replacing* `seq` *with its definition* }=

```
pure f >>= fun g =>
pure x >>= fun y =>
pure (g y)
```

={ `pure` *is a left identity of* `>>=` }=

```
pure f >>= fun g =>
pure (g x)
```

={ `pure` *is a left identity of* `>>=` }=

```
pure (f x)
```

And finally, to check that the ordering of pure operations doesn't matter:

```
seq u (fun () => pure x)
```

={ *Definition of* `seq` }=

```
u >>= fun f =>
pure x >>= fun y =>
pure (f y)
```

={ `pure` *is a left identity of* `>>=` }=

```
u >>= fun f =>
pure (f x)
```

={ *Clever replacement of one expression by an equivalent one that makes the rule match* }=

```
u >>= fun f =>
pure ((fun g => g x) f)
```

={ `pure` *is a left identity of* `>>=` }=

```
pure (fun g => g x) >>= fun h =>
u >>= fun f =>
pure (h f)
```

={ *Definition of* `seq` }=

```
seq (pure (fun f => f x)) (fun () => u)
```

This justifies a definition of `Monad` that extends `Applicative`, with a default definition of `seq`:

```
class Monad (m : Type → Type) extends Applicative m where
  bind : m α → (α → m β) → m β
  seq f x :=
    bind f fun g =>
    bind (x ()) fun y =>
    pure (g y)
```

`Applicative`'s own default definition of `map` means that every `Monad` instance automatically generates `Applicative` and `Functor` instances as well.

Additional Stipulations

In addition to adhering to the individual contracts associated with each type class, combined implementations `Functor`, `Applicative` and `Monad` should work equivalently to these default implementations. In other words, a type that provides both `Applicative` and `Monad` instances should not have an implementation of `seq` that works differently from the version that the `Monad` instance generates as a default implementation. This is important because polymorphic functions may be refactored to replace a use of `>>=` with an equivalent use of `<*>`, or a use of `<*>` with an equivalent use of `>>=`. This refactoring should not change the meaning of programs that use this code.

This rule explains why `Validate.andThen` should not be used to implement `bind` in a `Monad` instance. On its own, it obeys the monad contract. However, when it is used to implement `seq`, the behavior is not equivalent to `seq` itself. To see where they differ, take the example of two computations, both of which return errors. Start with an example of a case where two errors should be returned, one from validating a function (which could have just as well resulted from a prior argument to the function), and one from validating an argument:

```
def notFun : Validate String (Nat → String) :=
  .errors { head := "First error", tail := [] }

def notArg : Validate String Nat :=
  .errors { head := "Second error", tail := [] }
```

Combining them with the version of `<*>` from `Validate`'s `Applicative` instance results in both errors being reported to the user:

```
notFun <*> notArg
===>
match notFun with
| .ok g => g <$> notArg
| .errors errs =>
  match notArg with
  | .ok _ => .errors errs
  | .errors errs' => .errors (errs ++ errs')
===>
match notArg with
| .ok _ => .errors { head := "First error", tail := [] }
| .errors errs' => .errors ({ head := "First error", tail := [] } ++ errs')
===>
.errors ({ head := "First error", tail := [] } ++ { head := "Second error", tail
:= []})
===>
.errors { head := "First error", tail := ["Second error"]}
```

Using the version of `seq` that was implemented with `>>=`, here rewritten to `andThen`, results in only the first error being available:

```
seq notFun (fun () => notArg)
===>
notFun.andThen fun g =>
notArg.andThen fun y =>
pure (g y)
===>
match notFun with
| .errors errs => .errors errs
| .ok val =>
  (fun g =>
    notArg.andThen fun y =>
    pure (g y)) val
===>
.errors { head := "First error", tail := [] }
```

Alternatives

Recovery from Failure

`Validate` can also be used in situations where there is more than one way for input to be acceptable. For the input form `RawInput`, an alternative set of business rules that implement conventions from a legacy system might be the following:

1. All human users must provide a birth year that is four digits.
2. Users born prior to 1970 do not need to provide names, due to incomplete older records.
3. Users born after 1970 must provide names.
4. Companies should enter `"FIRM"` as their year of birth and provide a company name.

No particular provision is made for users born in 1970. It is expected that they will either give up, lie about their year of birth, or call. The company considers this an acceptable cost of doing business.

The following inductive type captures the values that can be produced from these stated rules:

```
abbrev NonEmptyString := {s : String // s ≠ ""}

inductive LegacyCheckedInput where
  | humanBefore1970 :
    (birthYear : {y : Nat // y > 999 ∧ y < 1970}) →
    String →
    LegacyCheckedInput
  | humanAfter1970 :
    (birthYear : {y : Nat // y > 1970}) →
    NonEmptyString →
    LegacyCheckedInput
  | company :
    NonEmptyString →
    LegacyCheckedInput
deriving Repr
```

A validator for these rules is more complicated, however, as it must address all three cases. While it can be written as a series of nested `if` expressions, it's easier to design the three cases independently and then combine them. This requires a means of recovering from failure while preserving error messages:

```
def Validate.orElse (a : Validate ε α) (b : Unit → Validate ε α) : Validate ε α
:=
  match a with
  | .ok x => .ok x
  | .errors errs1 =>
    match b () with
    | .ok x => .ok x
    | .errors errs2 => .errors (errs1 ++ errs2)
```

This pattern of recovery from failures is common enough that Lean has built-in syntax for it, attached to a type class named `OrElse`:

```
class OrElse (α : Type) where
  orElse : α → (Unit → α) → α
```

The expression `E1 <|> E2` is short for `OrElse.orElse E1 (fun () => E2)`. An instance of `OrElse` for `Validate` allows this syntax to be used for error recovery:

```
instance : OrElse (Validate ε α) where
  orElse := Validate.orElse
```

The validator for `LegacyCheckedInput` can be built from a validator for each constructor. The rules for a company state that the birth year should be the string `"FIRM"` and that the name should be non-empty. The constructor `LegacyCheckedInput.company`, however, has no representation of the birth year at all, so there's no easy way to carry it out using `<*>`. The key is to use a function with `<*>` that ignores its argument.

Checking that a Boolean condition holds without recording any evidence of this fact in a type can be accomplished with `checkThat`:

```
def checkThat (condition : Bool) (field : Field) (msg : String) : Validate
(Field × String) Unit :=
  if condition then pure () else reportError field msg
```

This definition of `checkCompany` uses `checkThat`, and then throws away the resulting `Unit` value:

```
def checkCompany (input : RawInput) : Validate (Field × String)
LegacyCheckedInput :=
  pure (fun () name => .company name) <*>
    checkThat (input.birthYear == "FIRM") "birth year" "FIRM if a company" <*>
    checkName input.name
```

However, this definition is quite noisy. It can be simplified in two ways. The first is to replace the first use of `<*>` with a specialized version that automatically ignores the value returned by the first argument, called `*>`. This operator is also controlled by a type class, called `SeqRight`, and `E1 *> E2` is syntactic sugar for `SeqRight.seqRight E1 (fun () => E2)`:

```
class SeqRight (f : Type → Type) where
  seqRight : f α → (Unit → f β) → f β
```

There is a default implementation of `seqRight` in terms of `seq`: `seqRight (a : f α) (b : Unit → f β) : f β := pure (fun _ x => x) <*> a <*> b ()`.

Using `seqRight`, `checkCompany` becomes simpler:

```
def checkCompany (input : RawInput) : Validate (Field × String)
LegacyCheckedInput :=
  checkThat (input.birthYear == "FIRM") "birth year" "FIRM if a company" *>
  pure .company <*> checkName input.name
```

One more simplification is possible. For every `Applicative`, `pure F <*> E` is equivalent to `f <$> E`. In other words, using `seq` to apply a function that was placed into the `Applicative` type using `pure` is overkill, and the function could have just been applied using `Functor.map`. This simplification yields:

```
def checkCompany (input : RawInput) : Validate (Field × String)
LegacyCheckedInput :=
  checkThat (input.birthYear == "FIRM") "birth year" "FIRM if a company" *>
  .company <$> checkName input.name
```

The remaining two constructors of `LegacyCheckedInput` use subtypes for their fields. A general-purpose tool for checking subtypes will make these easier to read:

```
def checkSubtype {α : Type} (v : α) (p : α → Prop) [Decidable (p v)] (err : ε) :
Validate ε {x : α // p x} :=
  if h : p v then
    pure ⟨v, h⟩
  else
    .errors { head := err, tail := [] }
```

In the function's argument list, it's important that the type class `[Decidable (p v)]` occur after the specification of the arguments `v` and `p`. Otherwise, it would refer to an additional set of automatic implicit arguments, rather than to the manually-provided values. The `Decidable` instance is what allows the proposition `p v` to be checked using `if`.

The two human cases do not need any additional tools:

```
def checkHumanBefore1970 (input : RawInput) : Validate (Field × String)
LegacyCheckedInput :=
  (checkYearIsNat input.birthYear).andThen fun y =>
    .humanBefore1970 <$>
      checkSubtype y (fun x => x > 999 ∧ x < 1970) ("birth year", "less than
1970") <*>
      pure input.name

def checkHumanAfter1970 (input : RawInput) : Validate (Field × String)
LegacyCheckedInput :=
  (checkYearIsNat input.birthYear).andThen fun y =>
    .humanAfter1970 <$>
      checkSubtype y (· > 1970) ("birth year", "greater than 1970") <*>
      checkName input.name
```

The validators for the three cases can be combined using `<|>`:

```
def checkLegacyInput (input : RawInput) : Validate (Field × String)
LegacyCheckedInput :=
  checkCompany input <|> checkHumanBefore1970 input <|> checkHumanAfter1970
input
```

The successful cases return constructors of `LegacyCheckedInput`, as expected:

```
#eval checkLegacyInput ⟨"Johnny's Troll Groomers", "FIRM"⟩
```

```
Validate.ok (LegacyCheckedInput.company "Johnny's Troll Groomers")
```

```
#eval checkLegacyInput ⟨"Johnny", "1963"⟩
```

```
Validate.ok (LegacyCheckedInput.humanBefore1970 1963 "Johnny")
```

```
#eval checkLegacyInput ⟨"", "1963"⟩
```

```
Validate.ok (LegacyCheckedInput.humanBefore1970 1963 "")
```

The worst possible input returns all the possible failures:

```
#eval checkLegacyInput ⟨"", "1970"⟩
```

```
Validate.errors
  { head := ("birth year", "FIRM if a company"),
    tail := [("name", "Required"),
             ("birth year", "less than 1970"),
             ("birth year", "greater than 1970"),
             ("name", "Required")] }
```

The `Alternative` Class

Many types support a notion of failure and recovery. The `Many` monad from the section on evaluating arithmetic expressions in a variety of monads is one such type, as is `Option`. Both support failure without providing a reason (unlike, say, `Except` and `Validate`, which require some indication of what went wrong).

The `Alternative` class describes applicative functors that have additional operators for failure and recovery:

```
class Alternative (f : Type → Type) extends Applicative f where
  failure : f α
  orElse : f α → (Unit → f α) → f α
```

Just as implementors of `Add α` get `HAdd α α α` instances for free, implementors of `Alternative` get `OrElse` instances for free:

```
instance [Alternative f] : OrElse (f α) where
  orElse := Alternative.orElse
```

The implementation of `Alternative` for `Option` keeps the first none- `none` argument:

```
instance : Alternative Option where
  failure := none
  orElse
    | some x, _ => some x
    | none, y => y ()
```

Similarly, the implementation for `Many` follows the general structure of `Many.union`, with minor differences due to the laziness-inducing `Unit` parameters being placed differently:

```
def Many.orElse : Many α → (Unit → Many α) → Many α
  | .none, ys => ys ()
  | .more x xs, ys => .more x (fun () => orElse (xs ()) ys)

instance : Alternative Many where
  failure := .none
  orElse := Many.orElse
```

Like other type classes, `Alternative` enables the definition of a variety of operations that work for *any* applicative functor that implements `Alternative`. One of the most important is `guard`, which causes `failure` when a decidable proposition is false:

```
def guard [Alternative f] (p : Prop) [Decidable p] : f Unit :=
  if p then
    pure ()
  else failure
```

It is very useful in monadic programs to terminate execution early. In `Many`, it can be used to filter out a whole branch of a search, as in the following program that computes all even divisors of a natural number:

```
def Many.countdown : Nat → Many Nat
  | 0 => .none
  | n + 1 => .more n (fun () => countdown n)

def evenDivisors (n : Nat) : Many Nat := do
  let k ← Many.countdown (n + 1)
  guard (k % 2 = 0)
  guard (n % k = 0)
  pure k
```

Running it on `20` yields the expected results:

```
#eval (evenDivisors 20).takeAll
```

```
[20, 10, 4, 2]
```

Exercises

Improve Validation Friendliness

The errors returned from `Validate` programs that use `<|>` can be difficult to read, because inclusion in the list of errors simply means that the error can be reached through *some* code path. A more structured error report can be used to guide the user through the process more accurately:

- Replace the `NonEmptyList` in `Validate.error` with a bare type variable, and then update the definitions of the `Applicative (Validate ε)` and `OrElse (Validate ε α)` instances to require only that there be an `Append ε` instance available.
- Define a function `Validate.mapErrors : Validate ε α → (ε → ε') → Validate ε' α` that transforms all the errors in a validation run.
- Using the datatype `TreeError` to represent errors, rewrite the legacy validation system to track its path through the three alternatives.
- Write a function `report : TreeError → String` that outputs a user-friendly view of the `TreeError` 's accumulated warnings and errors.

```
inductive TreeError where
  | field : Field → String → TreeError
  | path : String → TreeError → TreeError
  | both : TreeError → TreeError → TreeError

instance : Append TreeError where
  append := .both
```

Universes

In the interests of simplicity, this book has thus far papered over an important feature of Lean: *universes*. A universe is a type that classifies other types. Two of them are familiar: `Type` and `Prop`. `Type` classifies ordinary types, such as `Nat`, `String`, `Int → String × Char`, and `IO Unit`. `Prop` classifies propositions that may be true or false, such as `"nisse" = "elf"` or `3 > 2`. The type of `Prop` is `Type`:

```
#check Prop
```

```
Prop : Type
```

For technical reasons, more universes than these two are needed. In particular, `Type` cannot itself be a `Type`. This would allow a logical paradox to be constructed and undermine Lean's usefulness as a theorem prover.

The formal argument for this is known as *Girard's Paradox*. It is related to a better-known paradox known as *Russell's Paradox*, which was used to show that early versions of set theory were inconsistent. In these set theories, a set can be defined by a property. For example, one might have the set of all red things, the set of all fruit, the set of all natural numbers, or even the set of all sets. Given a set, one can ask whether a given element is contained in it. For instance, a bluebird is not contained in the set of all red things, but the set of all red things is contained in the set of all sets. Indeed, the set of all sets even contains itself.

What about the set of all sets that do not contain themselves? It contains the set of all red things, as the set of all red things is not itself red. It does not contain the set of all sets, because the set of all sets contains itself. But does it contain itself? If it does contain itself, then it cannot contain itself. But if it does not, then it must.

This is a contradiction, which demonstrates that something was wrong with the initial assumptions. In particular, allowing sets to be constructed by providing an arbitrary property is too powerful. Later versions of set theory restrict the formation of sets to remove the paradox.

A related paradox can be constructed in versions of dependent type theory that assign the type `Type` to `Type`. To ensure that Lean has consistent logical foundations and can be used as a tool for mathematics, `Type` needs to have some other type. This type is called `Type 1`:

```
#check Type
```

```
Type : Type 1
```

Similarly, `Type 1` is a `Type 2`, `Type 2` is a `Type 3`, `Type 3` is a `Type 4`, and so forth.

Function types occupy the smallest universe that can contain both the argument type and the return type. This means that `Nat → Nat` is a `Type`, `Type → Type` is a `Type 1`, and `Type 1 → Type 2` is a `Type 3`.

There is one exception to this rule. If the return type of a function is a `Prop`, then the whole function type is in `Prop`, even if the argument is in a larger universe such as `Type` or even `Type 1`. In particular, this means that predicates over values that have ordinary types are in `Prop`. For example, the type `(n : Nat) → n = n + 0` represents a function from a `Nat` to evidence that it is equal to itself plus zero. Even though `Nat` is in `Type`, this function type is in `Prop` due to this rule. Similarly, even though `Type` is in `Type 1`, the function type `Type → 2 + 2 = 4` is still in `Prop`.

User Defined Types

Structures and inductive datatypes can be declared to inhabit particular universes. Lean then checks whether each datatype avoids paradoxes by being in a universe that's large enough to prevent it from containing its own type. For instance, in the following declaration, `MyList` is declared to reside in `Type`, and so is its type argument `α`:

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

`MyList` itself is a `Type → Type`. This means that it cannot be used to contain actual types, because then its argument would be `Type`, which is a `Type 1`:

```
def myListOfNat : MyList Type :=
  .cons Nat .nil
```

```
application type mismatch
  MyList Type
argument
  Type
has type
  Type 1 : Type 2
but is expected to have type
  Type : Type 1
```

Updating `MyList` so that its argument is a `Type 1` results in a definition rejected by Lean:

```
inductive MyList (α : Type 1) : Type where
  | nil : MyList α
  | cons : α → MyList α → MyList α
```

```
invalid universe level in constructor 'MyList.cons', parameter has type
  α
at universe level
  2
which is not less than or equal to the inductive type's resulting universe level
  1
```

This error occurs because the argument to `cons` with type `α` is from a larger universe than `MyList`. Placing `MyList` itself in `Type 1` solves this issue, but at the cost of `MyList` now being itself inconvenient to use in contexts that expect a `Type`.

The specific rules that govern whether a datatype is allowed are somewhat complicated. Generally speaking, it's easiest to start with the datatype in the same universe as the largest of its arguments. Then, if Lean rejects the definition, increase its level by one, which will usually go through.

Universe Polymorphism

Defining a datatype in a specific universe can lead to code duplication. Placing `MyList` in `Type → Type` means that it can't be used for an actual list of types. Placing it in `Type 1 → Type 1` means that it can't be used for a list of lists of types. Rather than copy-pasting the datatype to create versions in `Type`, `Type 1`, `Type 2`, and so on, a feature called *universe polymorphism* can be used to write a single definition that can be instantiated in any of these universes.

Ordinary polymorphic types use variables to stand for types in a definition. This allows Lean to fill in the variables differently, which enables these definitions to be used with a variety of types. Similarly, universe polymorphism allows variables to stand for universes in a definition, enabling Lean to fill them in differently so that they can be used with a variety of universes. Just as type arguments are conventionally named with Greek letters, universe arguments are conventionally named `u`, `v`, and `w`.

This definition of `MyList` doesn't specify a particular universe level, but instead uses a variable `u` to stand for any level. If the resulting datatype is used with `Type`, then `u` is `0`, and if it's used with `Type 3`, then `u` is `3`:

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

With this definition, the same definition of `MyList` can be used to contain both actual natural numbers and the natural number type itself:

```
def myListOfNumbers : MyList Nat :=
  .cons 0 (.cons 1 .nil)

def myListOfNat : MyList Type :=
  .cons Nat .nil
```

It can even contain itself:

```
def myListOfList : MyList (Type → Type) :=
  .cons MyList .nil
```

It would seem that this would make it possible to write a logical paradox. After all, the whole point of the universe system is to rule out self-referential types. Behind the scenes, however, each occurrence of `MyList` is provided with a universe level argument. In essence, the universe-polymorphic definition of `MyList` created a *copy* of the datatype at each level, and the level argument selects which copy is to be used. These level arguments are written with a dot and curly braces, so `MyList.{0} : Type → Type`, `MyList.{1} : Type 1 → Type 1`, and `MyList.{2} : Type 2 → Type 2`.

Writing the levels explicitly, the prior example becomes:

```
def myListOfNumbers : MyList.{0} Nat :=
  .cons 0 (.cons 1 .nil)

def myListOfNat : MyList.{1} Type :=
  .cons Nat .nil

def myListOfList : MyList.{1} (Type → Type) :=
  .cons MyList.{0} .nil
```

When a universe-polymorphic definition takes multiple types as arguments, it's a good idea to give each argument its own level variable for maximum flexibility. For example, a version of `Sum` with a single level argument can be written as follows:

```
inductive Sum (α : Type u) (β : Type u) : Type u where
  | inl : α → Sum α β
  | inr : β → Sum α β
```

This definition can be used at multiple levels:

```
def stringOrNat : Sum String Nat := .inl "hello"

def typeOrType : Sum Type Type := .inr Nat
```

However, it requires that both arguments be in the same universe:

```
def stringOrType : Sum String Type := .inr Nat
```

```
application type mismatch
  Sum String Type
argument
  Type
has type
  Type 1 : Type 2
but is expected to have type
  Type : Type 1
```

This datatype can be made more flexible by using different variables for the two type arguments' universe levels, and then declaring that the resulting datatype is in the largest of the two:

```
inductive Sum (α : Type u) (β : Type v) : Type (max u v) where
  | inl : α → Sum α β
  | inr : β → Sum α β
```

This allows `Sum` to be used with arguments from different universes:

```
def stringOrType : Sum String Type := .inr Nat
```

In positions where Lean expects a universe level, any of the following are allowed:

- A concrete level, like `0` or `1`
- A variable that stands for a level, such as `u` or `v`
- The maximum of two levels, written as `max` applied to the levels
- A level increase, written with `+ 1`

Writing Universe-Polymorphic Definitions

Until now, every datatype defined in this book has been in `Type`, the smallest universe of data. When presenting polymorphic datatypes from the Lean standard library, such as `List` and `Sum`, this book created non-universe-polymorphic versions of them. The real versions use universe polymorphism to enable code re-use between type-level and non-type-level programs.

There are a few general guidelines to follow when writing universe-polymorphic types. First off, independent type arguments should have different universe variables, which enables the polymorphic definition to be used with a wider variety of arguments, increasing the potential for code reuse. Secondly, the whole type is itself typically either in the maximum of all the universe variables, or one greater than this maximum. Try the smaller of the two first. Finally, it's a good idea to put the new type in as small of a universe as possible, which allows it to be used more flexibly in other contexts. Non-polymorphic types, such as `Nat` and `String`, can be placed directly in `Type 0`.

`Prop` and Polymorphism

Just as `Type`, `Type 1`, and so on describe types that classify programs and data, `Prop` classifies logical propositions. A type in `Prop` describes what counts as convincing evidence for the truth of a statement. Propositions are like ordinary types in many ways: they can be declared inductively, they can have constructors, and functions can take propositions as arguments. However, unlike datatypes, it typically doesn't matter *which* evidence is provided for the truth of a statement, only *that* evidence is provided. On the other hand, it is very important that a program not only return a `Nat`, but that it's the *correct* `Nat`.

`Prop` is at the bottom of the universe hierarchy, and the type of `Prop` is `Type`. This means that `Prop` is a suitable argument to provide to `List`, for the same reason that `Nat` is. Lists of propositions have type `List Prop`:

```
def someTruePropositions : List Prop := [
  1 + 1 = 2,
  "Hello, " ++ "world!" = "Hello, world!"
]
```

Filling out the universe argument explicitly demonstrates that `Prop` is a `Type`:

```
def someTruePropositions : List.{0} Prop := [
  1 + 1 = 2,
  "Hello, " ++ "world!" = "Hello, world!"
]
```

Behind the scenes, `Prop` and `Type` are united into a single hierarchy called `Sort`. `Prop` is the same as `Sort 0`, `Type 0` is `Sort 1`, `Type 1` is `Sort 2`, and so forth. In fact, `Type u` is the same as `Sort (u+1)`. When writing programs with Lean, this is typically not relevant, but it may occur in error messages from time to time, and it explains the name of the `CoeSort` class. Additionally, having `Prop` as `Sort 0` allows one more universe operator to become useful. The universe level `imax u v` is `0` when `v` is `0`, or the larger of `u` or `v` otherwise. Together with `Sort`, this allows the special rule for functions that return `Prop`s to be used when writing code that should be as portable as possible between `Prop` and `Type` universes.

Polymorphism in Practice

In the remainder of the book, definitions of polymorphic datatypes, structures, and classes will use universe polymorphism in order to be consistent with the Lean standard library. This will enable the complete presentation of the `Functor`, `Applicative`, and `Monad` classes to be completely consistent with their actual definitions.

The Complete Definitions

Now that all the relevant language features have been presented, this section describes the complete, honest definitions of `Functor`, `Applicative`, and `Monad` as they occur in the Lean standard library. For the sake of understanding, no details are omitted.

Functor

The complete definition of the `Functor` class makes use of universe polymorphism and a default method implementation:

```
class Functor (f : Type u → Type v) : Type (max (u+1) v) where
  map : {α β : Type u} → (α → β) → f α → f β
  mapConst : {α β : Type u} → α → f β → f α :=
    Function.comp map (Function.const _)
```

In this definition, `Function.comp` is function composition, which is typically written with the `∘` operator. `Function.const` is the *constant function*, which is a two-argument function that ignores its second argument. Applying this function to only one argument produces a function that always returns the same value, which is useful when an API demands a function but a program doesn't need to compute different results for different arguments. A simple version of `Function.const` can be written as follows:

```
def simpleConst  (x : α) (_ : β) : α := x
```

Using it with one argument as the function argument to `List.map` demonstrates its utility:

```
#eval [1, 2, 3].map (simpleConst "same")
```

```
["same", "same", "same"]
```

The actual function has the following signature:

```
Function.const.{u, v} {α : Sort u} (β : Sort v) (a : α) : β → α
```

Here, the type argument `β` is an explicit argument, so the default definition of `Functor.mapConst` provides an `_` argument that instructs Lean to find a unique type to pass to `Function.const` that would cause the program to type check. `(Function.comp map (Function.const _) : α → f β → f α)` is equivalent to `fun (x : α) (y : f β) => map` `(fun _ => x) y` .

The `Functor` type class inhabits a universe that is the greater of `u+1` and `v` . Here, `u` is the level of universes accepted as arguments to `f` , while `v` is the universe returned by `f` . To see why the structure that implements the `Functor` type class must be in a universe that's larger than `u` , begin with a simplified definition of the class:

```
class Functor (f : Type u → Type v) : Type (max (u+1) v) where
  map : {α β : Type u} → (α → β) → f α → f β
```

This type class's structure type is equivalent to the following inductive type:

```
inductive Functor (f : Type u → Type v) : Type (max (u+1) v) where
  | mk : ({α β : Type u} → (α → β) → f α → f β) → Functor f
```

The implementation of the `map` method that is passed as an argument to `Functor.mk` contains a function that takes two types in `Type u` as arguments. This means that the type of the function itself is in `Type (u+1)` , so `Functor` must also be at a level that is at least `u+1` . Similarly, other arguments to the function have a type built by applying `f` , so it must also have a level that is at least `v` . All the type classes in this section share this property.

Applicative

The `Applicative` type class is actually built from a number of smaller classes that each contain some of the relevant methods. The first are `Pure` and `Seq` , which contain `pure` and `seq` respectively:

```
class Pure (f : Type u → Type v) : Type (max (u+1) v) where
  pure {α : Type u} : α → f α

class Seq (f : Type u → Type v) : Type (max (u+1) v) where
  seq : {α β : Type u} → f (α → β) → (Unit → f α) → f β
```

In addition to these, `Applicative` also depends on `SeqRight` and an analogous `SeqLeft` class:

```
class SeqRight (f : Type u → Type v) : Type (max (u+1) v) where
  seqRight : {α β : Type u} → f α → (Unit → f β) → f β

class SeqLeft (f : Type u → Type v) : Type (max (u+1) v) where
  seqLeft : {α β : Type u} → f α → (Unit → f β) → f α
```

The `seqRight` function, which was introduced in the section about alternatives and validation, is easiest to understand from the perspective of effects. `E1 *> E2`, which desugars to `SeqRight.seqRight E1 (fun () => E2)`, can be understood as first executing `E1`, and then `E2`, resulting only in `E2`'s result. Effects from `E1` may result in `E2` not being run, or being run multiple times. Indeed, if `f` has a `Monad` instance, then `E1 *> E2` is equivalent to `do let _ ← E1; E2`, but `seqRight` can be used with types like `Validate` that are not monads.

Its cousin `seqLeft` is very similar, except the leftmost expression's value is returned. `E1 <* E2` desugars to `SeqLeft.seqLeft E1 (fun () => E2)`. `SeqLeft.seqLeft` has type `f α → (Unit → f β) → f α`, which is identical to that of `seqRight` except for the fact that it returns `f α`. `E1 <* E2` can be understood as a program that first executes `E1`, and then `E2`, returning the original result for `E1`. If `f` has a `Monad` instance, then `E1 <* E2` is equivalent to `do let x ← E1; _ ← E2; pure x`. Generally speaking, `seqLeft` is useful for specifying extra conditions on a value in a validation or parser-like workflow without changing the value itself.

The definition of `Applicative` extends all these classes, along with `Functor`:

```
class Applicative (f : Type u → Type v) extends Functor f, Pure f, Seq f,
SeqLeft f, SeqRight f where
  map      := fun x y => Seq.seq (pure x) fun _ => y
  seqLeft  := fun a b => Seq.seq (Functor.map (Function.const _) a) b
  seqRight := fun a b => Seq.seq (Functor.map (Function.const _ id) a) b
```

A complete definition of `Applicative` requires only definitions for `pure` and `seq`. This is because there are default definitions for all of the methods from `Functor`, `SeqLeft`, and `SeqRight`. The `mapConst` method of `Functor` has its own default implementation in terms of `Functor.map`. These default implementations should only be overridden with new functions that are behaviorally equivalent, but more efficient. The default implementations should be seen as specifications for correctness as well as automatically-created code.

The default implementation for `seqLeft` is very compact. Replacing some of the names with their syntactic sugar or their definitions can provide another view on it, so:

```
fun a b => Seq.seq (Functor.map (Function.const _) a) b
```

becomes

```
fun a b => Seq.seq ((fun x _ => x) <$> a) b
```

How should `(fun x _ => x) <$> a` be understood? Here, `a` has type `f α`, and `f` is a functor. If `f` is `List`, then `(fun x _ => x) <$> [1, 2, 3]` evaluates to `[fun _ => 1, fun`

`_ => 2, fun _ => 3]`. If `f` is `Option`, then `(fun x _ => x) <$> some "hello"` evaluates to `some (fun _ => "hello")`. In each case, the values in the functor are replaced by functions that return the original value, ignoring their argument. When combined with `seq`, this function discards the values from `seq`'s second argument.

The default implementation for `seqRight` is very similar, except `const` has an additional argument `id`. This definition can be understood similarly, by first introducing some standard syntactic sugar and then replacing some names with their definitions:

```
fun a b => Seq.seq (Functor.map (Function.const _ id) a) b
===>
fun a b => Seq.seq ((fun _ => id) <$> a) b
===>
fun a b => Seq.seq ((fun _ => fun x => x) <$> a) b
===>
fun a b => Seq.seq ((fun _ x => x) <$> a) b
```

How should `(fun _ x => x) <$> a` be understood? Once again, examples are useful. `(fun _ x => x) <$> [1, 2, 3]` is equivalent to `[fun x => x, fun x => x, fun x => x]`, and `(fun _ x => x) <$> some "hello"` is equivalent to `some (fun x => x)`. In other words, `(fun _ x => x) <$> a` preserves the overall shape of `a`, but each value is replaced by the identity function. From the perspective of effects, the side effects of `a` occur, but the values are thrown out when it is used with `seq`.

Monad

Just as the constituent operations of `Applicative` are split into their own type classes, `Bind` has its own class as well:

```
class Bind (m : Type u → Type v) where
  bind : {α β : Type u} → m α → (α → m β) → m β
```

`Monad` extends `Applicative` with `Bind`:

```
class Monad (m : Type u → Type v) : Type (max (u+1) v) extends Applicative m,
Bind m  where
  map      f x := bind x (Function.comp pure f)
  seq      f x := bind f fun y => Functor.map y (x ())
  seqLeft  x y := bind x fun a => bind (y ()) (fun _ => pure a)
  seqRight x y := bind x fun _ => y ()
```

Tracing the collection of inherited methods and default methods from the entire hierarchy shows that a `Monad` instance requires only implementations of `bind` and `pure`. In other words, `Monad` instances automatically yield implementations of `seq`, `seqLeft`, `seqRight`, `map`, and `mapConst`. From the perspective of API boundaries, any type with a `Monad` instance gets instances for `Bind`, `Pure`, `Seq`, `Functor`, `SeqLeft`, and `SeqRight`.

Exercises

1. Understand the default implementations of `map`, `seq`, `seqLeft`, and `seqRight` in `Monad` by working through examples such as `Option` and `Except`. In other words, substitute their definitions for `bind` and `pure` into the default definitions, and simplify them to recover the versions `map`, `seq`, `seqLeft`, and `seqRight` that would be written by hand.
2. On paper or in a text file, prove to yourself that the default implementations of `map` and `seq` satisfy the contracts for `Functor` and `Applicative`. In this argument, you're allowed to use the rules from the `Monad` contract as well as ordinary expression evaluation.

Summary

Type Classes and Structures

Behind the scenes, type classes are represented by structures. Defining a class defines a structure, and additionally creates an empty table of instances. Defining an instance creates a value that either has the structure as its type or is a function that can return the structure, and additionally adds an entry to the table. Instance search consists of constructing an instance by consulting the instance tables. Both structures and classes may provide default values for fields (which are default implementations of methods).

Structures and Inheritance

Structures may inherit from other structures. Behind the scenes, a structure that inherits from another structure contains an instance of the original structure as a field. In other words, inheritance is implemented with composition. When multiple inheritance is used, only the unique fields from the additional parent structures are used to avoid a diamond problem, and the functions that would normally extract the parent value are instead organized to construct one. Record dot notation takes structure inheritance into account.

Because type classes are just structures with some additional automation applied, all of these features are available in type classes. Together with default methods, this can be used to create a fine-grained hierarchy of interfaces that nonetheless does not impose a large burden on clients, because the small classes that the large classes inherit from can be automatically implemented.

Applicative Functors

An applicative functor is a functor with two additional operations:

- `pure` , which is the same operator as that for `Monad`
- `seq` , which allows a function to be applied in the context of the functor.

While monads can represent arbitrary programs with control flow, applicative functors can only run function arguments from left to right. Because they are less powerful, they provide less control to programs written against the interface, while the implementor of the method has a greater degree of freedom. Some useful types can implement `Applicative` but not `Monad`.

In fact, the type classes `Functor`, `Applicative`, and `Monad` form a hierarchy of power. Moving up the hierarchy, from `Functor` towards `Monad`, allows more powerful programs to be written, but fewer types implement the more powerful classes. Polymorphic programs should be written to use as weak of an abstraction as possible, while datatypes should be given instances that are as powerful as possible. This maximizes code re-use. The more powerful type classes extend the less powerful ones, which means that an implementation of `Monad` provides implementations of `Functor` and `Applicative` for free.

Each class has a set of methods to be implemented and a corresponding contract that specifies additional rules for the methods. Programs that are written against these interfaces expect that the additional rules are followed, and may be buggy if they are not. The default implementations of `Functor`'s methods in terms of `Applicative`'s, and of `Applicative`'s in terms of `Monad`'s, will obey these rules.

Universes

To allow Lean to be used as both a programming language and a theorem prover, some restrictions on the language are necessary. This includes restrictions on recursive functions that ensure that they all either terminate or are marked as `partial` and written to return types that are not uninhabited. Additionally, it must be impossible to represent certain kinds of logical paradoxes as types.

One of the restrictions that rules out certain paradoxes is that every type is assigned to a *universe*. Universes are types such as `Prop`, `Type`, `Type 1`, `Type 2`, and so forth. These types describe other types—just as `0` and `17` are described by `Nat`, `Nat` is itself described by `Type`, and `Type` is described by `Type 1`. The type of functions that take a type as an argument must be a larger universe than the argument's universe.

Because each declared datatype has a universe, writing code that uses types like data would quickly become annoying, requiring each polymorphic type to be copy-pasted to take arguments from `Type 1`. A feature called *universe polymorphism* allows Lean programs and datatypes to take universe levels as arguments, just as ordinary polymorphism allows programs to take types as arguments. Generally speaking, Lean libraries should use universe polymorphism when implementing libraries of polymorphic operations.

Monad Transformers

A monad is a way to encode some collection of side effects in a pure language. Different monads provide different effects, such as state and error handling. Many monads even provide useful effects that aren't available in most languages, such as nondeterministic searches, readers, and even continuations.

A typical application has a core set of easily testable functions written without monads paired with an outer wrapper that uses a monad to encode the necessary application logic. These monads are constructed from well-known components. For example:

- Mutable state is encoded with a function parameter and a return value that have the same type
- Error handling is encoded by having a return type that is similar to `Except`, with constructors for success and failure
- Logging is encoded by pairing the return value with the log

Writing each monad by hand is tedious, however, involving boilerplate definitions of the various type classes. Each of these components can also be extracted to a definition that modifies some other monad to add an additional effect. Such a definition is called a *monad transformer*. A concrete monad can be build from a collection of monad transformers, which enables much more code re-use.

Combining IO and Reader

One case where a reader monad can be useful is when there is some notion of the "current configuration" of the application that is passed through many recursive calls. An example of such a program is `tree`, which recursively prints the files in the current directory and its subdirectories, indicating their tree structure using characters. The version of `tree` in this chapter, called `doug` after the mighty Douglas Fir tree that adorns the west coast of North America, provides the option of Unicode box-drawing characters or their ASCII equivalents when indicating directory structure.

For example, the following commands create a directory structure and some empty files in a directory called `doug-demo`:

```
$ cd doug-demo
$ mkdir -p a/b/c
$ mkdir -p a/d
$ mkdir -p a/e/f
$ touch a/b/hello
$ touch a/d/another-file
$ touch a/e/still-another-file-again
```

Running `doug` results in the following:

```
$ doug
├── doug-demo/
│   ├── a/
│   │   ├── b/
│   │   │   ├── c/
│   │   │   ├── hello
│   │   ├── d/
│   │   │   ├── another-file
│   │   ├── e/
│   │   │   ├── still-another-file-again
│   │   │   ├── f/
│   │   │   │
```

Implementation

Internally, `doug` passes a configuration value downwards as it recursively traverses the directory structure. This configuration contains two fields: `useASCII` determines whether to use Unicode box-drawing characters or ASCII vertical line and dash characters to indicate structure, and `currentPrefix` contains a string to prepend to each line of output. As the current directory deepens, the prefix string accumulates indicators of being in a directory.

The configuration is a structure:

```
structure Config where
  useASCII : Bool := false
  currentPrefix : String := ""
```

This structure has default definitions for both fields. The default `Config` uses Unicode display with no prefix.

Users who invoke `doug` will need to be able to provide command-line arguments. The usage information is as follows:

```
def usage : String :=
  "Usage: doug [--ascii]
Options:
\t--ascii\tUse ASCII characters to display the directory structure"
```

Accordingly, a configuration can be constructed by examining a list of command-line arguments:

```
def configFromArgs : List String → Option Config
  | [] => some {} -- both fields default
  | ["--ascii"] => some {useASCII := true}
  | _ => none
```

The `main` function is a wrapper around an inner worker, called `dirTree`, that shows the contents of a directory using a configuration. Before calling `dirTree`, `main` is responsible for processing command-line arguments. It must also return the appropriate exit code to the operating system:

```
def main (args : List String) : IO UInt32 := do
  match configFromArgs args with
  | some config =>
    dirTree config (← IO.currentDir)
    pure 0
  | none =>
    IO.eprintln s!"Didn't understand argument(s) {" ".separate args}\n"
    IO.eprintln usage
    pure 1
```

`IO.eprintln` is a version of `IO.println` that outputs to standard error.

Not all paths should be shown in the directory tree. In particular, files named `.` or `..` should be skipped, as they are actually features used for navigation rather than files *per se*. Of those files that should be shown, there are two kinds: ordinary files and directories:

```
inductive Entry where
  | file : String → Entry
  | dir : String → Entry
```

To determine whether a file should be shown, along with which kind of entry it is, `doug` uses `toEntry`:

```
def toEntry (path : System.FilePath) : IO (Option Entry) := do
  match path.components.getLast? with
  | none => pure (some (.dir ""))
  | some "." | some ".." => pure none
  | some name =>
    pure (some (if (← path.isDir) then .dir name else .file name))
```

`System.FilePath.components` converts a path into a list of path components, splitting the name at directory separators. If there is no last component, then the path is the root directory. If the last component is a special navigation file (`.` or `..`), then the file should be excluded. Otherwise, directories and files are wrapped in the corresponding constructors.

Lean's logic has no way to know that directory trees are finite. Indeed, some systems allow the construction of circular directory structures. Thus, `dirTree` is declared `partial`:

```
partial def dirTree (cfg : Config) (path : System.FilePath) : IO Unit := do
  match ← toEntry path with
  | none => pure ()
  | some (.file name) => showFileName cfg name
  | some (.dir name) =>
    showDirName cfg name
    let contents ← path.readDir
    let newConfig := cfg.inDirectory
    doList contents.toList fun d =>
      dirTree newConfig d.path
```

The call to `toEntry` is a nested action—the parentheses are optional in positions where the arrow couldn't have any other meaning, such as `match`. When the filename doesn't correspond to an entry in the tree (e.g. because it is `..`), `dirTree` does nothing. When the filename points to an ordinary file, `dirTree` calls a helper to show it with the current configuration. When the filename points to a directory, it is shown with a helper, and then its contents are recursively shown in a new configuration in which the prefix has been extended to account for being in a new directory.

Showing the names of files and directories is achieved with `showFileName` and `showDirName`:

```
def showFileName (cfg : Config) (file : String) : IO Unit := do
  IO.println (cfg.fileName file)

def showDirName (cfg : Config) (dir : String) : IO Unit := do
  IO.println (cfg.dirName dir)
```

Both of these helpers delegate to functions on `Config` that take the ASCII vs Unicode setting into account:

```
def Config.preFile (cfg : Config) :=
  if cfg.useASCII then "|--" else "├──"

def Config.preDir (cfg : Config) :=
  if cfg.useASCII then "|  " else "│  "

def Config.fileName (cfg : Config) (file : String) : String :=
  s!"{cfg.currentPrefix}{cfg.preFile} {file}"

def Config.dirName (cfg : Config) (dir : String) : String :=
  s!"{cfg.currentPrefix}{cfg.preFile} {dir}/"
```

Similarly, `Config.inDirectory` extends the prefix with a directory marker:

```
def Config.inDirectory (cfg : Config) : Config :=
  {cfg with currentPrefix := cfg.preDir ++ " " ++ cfg.currentPrefix}
```

Iterating an IO action over a list of directory contents is achieved using `doList`. Because `doList` carries out all the actions in a list and does not base control-flow decisions on the values returned by any of the actions, the full power of `Monad` is not necessary, and it will work for any `Applicative`:

```
def doList [Applicative f] : List α → (α → f Unit) → f Unit
  | [], _ => pure ()
  | x :: xs, action =>
    action x *>
    doList xs action
```

Using a Custom Monad

While this implementation of `doug` works, manually passing the configuration around is verbose and error-prone. The type system will not catch it if the wrong configuration is passed downwards, for instance. A reader effect ensures that the same configuration is passed to all recursive calls, unless it is manually overridden, and it helps make the code less verbose.

To create a version of `IO` that is also a reader of `Config`, first define the type and its `Monad` instance, following the recipe from the evaluator example:

```
def ConfigIO (α : Type) : Type :=
  Config → IO α

instance : Monad ConfigIO where
  pure x := fun _ => pure x
  bind result next := fun cfg => do
    let v ← result cfg
    next v cfg
```

The difference between this `Monad` instance and the one for `Reader` is that this one uses `do`-notation in the `IO` monad as the body of the function that `bind` returns, rather than applying `next` directly to the value returned from `result`. Any `IO` effects performed by `result` must occur before `next` is invoked, which is ensured by the `IO` monad's `bind` operator. `ConfigIO` is not universe polymorphic because the underlying `IO` type is also not universe polymorphic.

Running a `ConfigIO` action involves transforming it into an `IO` action by providing it with a configuration:

```
def ConfigIO.run (action : ConfigIO α) (cfg : Config) : IO α :=
  action cfg
```

This function is not really necessary, as a caller could simply provide the configuration directly. However, naming the operation can make it easier to see which parts of the code are intended to run in which monad.

The next step is to define a means of accessing the current configuration as part of `ConfigIO`:

```
def currentConfig : ConfigIO Config :=
  fun cfg => pure cfg
```

This is just like `read` from the evaluator example, except it uses `IO`'s `pure` to return its value rather than doing so directly. Because entering a directory modifies the current configuration for the scope of a recursive call, it will be necessary to have a way to override a configuration:

```
def locally (change : Config → Config) (action : ConfigIO α) : ConfigIO α :=
  fun cfg => action (change cfg)
```

Much of the code used in `doug` has no need for configurations, and `doug` calls ordinary Lean `IO` actions from the standard library that certainly don't need a `Config`. Ordinary `IO` actions can be run using `runIO`, which ignores the configuration argument:

```
def runIO (action : IO α) : ConfigIO α :=
  fun _ => action
```

With these components, `showFileName` and `showDirName` can be updated to take their configuration arguments implicitly through the `ConfigIO` monad. They use nested actions to retrieve the configuration, and `runIO` to actually execute the call to `IO.println`:

```
def showFileName (file : String) : ConfigIO Unit := do
  runIO (IO.println ((← currentConfig).fileName file))

def showDirName (dir : String) : ConfigIO Unit := do
  runIO (IO.println ((← currentConfig).dirName dir))
```

In the new version of `dirTree`, the calls to `toEntry` and `System.FilePath.readDir` are wrapped in `runIO`. Additionally, instead of building a new configuration and then requiring the programmer to keep track of which one to pass to recursive calls, it uses `locally` to naturally delimit the modified configuration to only a small region of the program, in which it is the *only* valid configuration:

```
partial def dirTree (path : System.FilePath) : ConfigIO Unit := do
  match ← runIO (toEntry path) with
    | none => pure ()
    | some (.file name) => showFileName name
    | some (.dir name) =>
      showDirName name
      let contents ← runIO path.readDir
      locally (·.inDirectory)
        (doList contents.toList fun d =>
          dirTree d.path)
```

The new version of `main` uses `ConfigIO.run` to invoke `dirTree` with the initial configuration:

```
def main (args : List String) : IO UInt32 := do
    match configFromArgs args with
    | some config =>
      (dirTree (← IO.currentDir)).run config
      pure 0
    | none =>
      IO.eprintln s!"Didn't understand argument(s) {" ".separate args}\n"
      IO.eprintln usage
      pure 1
```

This custom monad has a number of advantages over passing configurations manually:

1. It is easier to ensure that configurations are passed down unchanged, except when changes are desired
2. The concern of passing the configuration onwards is more clearly separated from the concern of printing directory contents
3. As the program grows, there will be more and more intermediate layers that do nothing with configurations except propagate them, and these layers don't need to be rewritten as the configuration logic changes

However, there are also some clear downsides:

1. As the program evolves and the monad requires more features, each of the basic operators such as `locally` and `currentConfig` will need to be updated
2. Wrapping ordinary `IO` actions in `runIO` is noisy and distracts from the flow of the program
3. Writing monads instances by hand is repetitive, and the technique for adding a reader effect to another monad is a design pattern that requires documentation and communication overhead

Using a technique called *monad transformers*, all of these downsides can be addressed. A monad transformer takes a monad as an argument and returns a new monad. Monad transformers consist of:

1. A definition of the transformer itself, which is typically a function from types to types
2. A `Monad` instance that assumes the inner type is already a monad
3. An operator to "lift" an action from the inner monad to the transformed monad, akin to `runIO`

Adding a Reader to Any Monad

Adding a reader effect to `IO` was accomplished in `ConfigIO` by wrapping `IO α` in a function type. The Lean standard library contains a function that can do this to *any* polymorphic type, called `ReaderT`:

```
def ReaderT (ρ : Type u) (m : Type u → Type v) (α : Type u) : Type (max u v) :=
  ρ → m α
```

Its arguments are as follows:

- `ρ` is the environment that is accessible to the reader

- `m` is the monad that is being transformed, such as `IO`
- `α` is the type of values being returned by the monadic computation Both `α` and `ρ` are in the same universe because the operator that retrieves the environment in the monad will have type `m ρ`.

With `ReaderT`, `ConfigIO` becomes:

```
abbrev ConfigIO (α : Type) : Type := ReaderT Config IO α
```

It is an `abbrev` because `ReaderT` has many useful features defined in the standard library that a non-reducible definition would hide. Rather than taking responsibility for making these work directly for `ConfigIO`, it's easier to simply have `ConfigIO` behave identically to `ReaderT Config IO`.

The manually-written `currentConfig` obtained the environment out of the reader. This effect can be defined in a generic form for all uses of `ReaderT`, under the name `read`:

```
def read [Monad m] : ReaderT ρ m ρ :=
   fun env => pure env
```

However, not every monad that provides a reader effect is built with `ReaderT`. The type class `MonadReader` allows any monad to provide a `read` operator:

```
class MonadReader (ρ : outParam (Type u)) (m : Type u → Type v) : Type (max (u +
1) v) where
  read : m ρ

instance [Monad m] : MonadReader ρ (ReaderT ρ m) where
  read := fun env => pure env

export MonadReader (read)
```

The type `ρ` is an output parameter because any given monad typically only provides a single type of environment through a reader, so automatically selecting it when the monad is known makes programs more convenient to write.

The `Monad` instance for `ReaderT` is essentially the same as the `Monad` instance for `ConfigIO`, except `IO` has been replaced by some arbitrary monad argument `m`:

```
instance [Monad m] : Monad (ReaderT ρ m) where
  pure x := fun _ => pure x
  bind result next := fun env => do
    let v ← result env
    next v env
```

The next step is to eliminate uses of `runIO`. When Lean encounters a mismatch in monad types, it automatically attempts to use a type class called `MonadLift` to transform the actual monad into the expected monad. This process is similar to the use of coercions. `MonadLift` is defined as follows:

```
class MonadLift (m : Type u → Type v) (n : Type u → Type w) where
  monadLift : {α : Type u} → m α → n α
```

The method `monadLift` translates from the monad `m` to the monad `n`. The process is called "lifting" because it takes an action in the embedded monad and makes it into an action in the surrounding monad. In this case, it will be used to "lift" from `IO` to `ReaderT Config IO`, though the instance works for *any* inner monad `m`:

```
instance : MonadLift m (ReaderT ρ m) where
  monadLift action := fun _ => action
```

The implementation of `monadLift` is very similar to that of `runIO`. Indeed, it is enough to define `showFileName` and `showDirName` without using `runIO`:

```
def showFileName (file : String) : ConfigIO Unit := do
  IO.println s!"{(← read).currentPrefix} {file}"

def showDirName (dir : String) : ConfigIO Unit := do
  IO.println s!"{(← read).currentPrefix} {dir}/"
```

One final operation from the original `ConfigIO` remains to be translated to a use of `ReaderT`: `locally`. The definition can be translated directly to `ReaderT`, but the Lean standard library provides a more general version. The standard version is called `withReader`, and it is part of a type class called `MonadWithReader`:

```
class MonadWithReader (ρ : outParam (Type u)) (m : Type u → Type v) where
  withReader {α : Type u} : (ρ → ρ) → m α → m α
```

Just as in `MonadReader`, the environment `ρ` is an `outParam`. The `withReader` operation is exported, so that it doesn't need to be written with the type class name before it:

```
export MonadWithReader (withReader)
```

The instance for `ReaderT` is essentially the same as the definition of `locally`:

```
instance : MonadWithReader ρ (ReaderT ρ m) where
  withReader change action :=
    fun cfg => action (change cfg)
```

With these definitions in place, the new version of `dirTree` can be written:

```
partial def dirTree (path : System.FilePath) : ConfigIO Unit := do
  match ← toEntry path with
    | none => pure ()
    | some (.file name) => showFileName name
    | some (.dir name) =>
      showDirName name
      let contents ← path.readDir
      withReader (·.inDirectory)
        (doList contents.toList fun d =>
          dirTree d.path)
```

Aside from replacing `locally` with `withReader`, it is the same as before.

Replacing the custom `ConfigIO` type with `ReaderT` did not save a large number of lines of code in this section. However, rewriting the code using components from the standard library does have long-term benefits. First, readers who know about `ReaderT` don't need to take time to understand the `Monad` instance for `ConfigIO`, working backwards to the meaning of monad itself. Instead, they can be confident in their initial understanding. Next, adding further effects to the monad (such as a state effect to count the files in each directory and display a count at the end) requires far fewer changes to the code, because the monad transformers and `MonadLift` instances provided in the library work well together. Finally, using a set of type classes included in the standard library, polymorphic code can be written in such a way that it can work with a variety of monads without having to care about details like the order in which the monad transformers were applied. Just as some functions work in any monad, others can work in any monad that provides a certain type of state, or a certain type of exceptions, without having to specifically describe the *way* in which a particular concrete monad provides the state or exceptions.

Exercises

Controlling the Display of Dotfiles

Files whose names begin with a dot character (`'.'`) typically represent files that should usually be hidden, such as source-control metadata and configuration files. Modify `doug` with an option to show or hide filenames that begin with a dot. This option should be controlled with a `-a` command-line option.

Starting Directory as Argument

Modify `doug` so that it takes a starting directory as an additional command-line argument.

A Monad Construction Kit

`ReaderT` is far from the only useful monad transformer. This section describes a number of additional transformers. Each monad transformer consists of the following:

1. A definition or datatype `T` that takes a monad as an argument. It should have a type like `(Type u → Type v) → Type u → Type v`, though it may accept additional arguments prior to the monad.
2. A `Monad` instance for `T m` that relies on an instance of `Monad m`. This enables the transformed monad to be used as a monad.
3. A `MonadLift` instance that translates actions of type `m α` into actions of type `T m α`, for arbitrary monads `m`. This enables actions from the underlying monad to be used in the transformed monad.

Furthermore, the `Monad` instance for the transformer should obey the contract for `Monad`, at least if the underlying `Monad` instance does. In addition, `monadLift (pure x)` should be equivalent to `pure x` in the transformed monad, and `monadLift` should distribute over `bind` so that `monadLift (x >>= f)` is the same as `monadLift x >>= fun y => monadLift (f y)`.

Many monad transformers additionally define type classes in the style of `MonadReader` that describe the actual effects available in the monad. This can provide more flexibility: it allows programs to be written that rely only on an interface, and don't constrain the underlying monad to be implemented by a given transformer. The type classes are a way for programs to express their requirements, and monad transformers are a convenient way to meet these requirements.

Failure with `OptionT`

Failure, represented by the `Option` monad, and exceptions, represented by the `Except` monad, both have corresponding transformers. In the case of `Option`, failure can be added to a monad by having it contain values of type `Option α` where it would otherwise contain values of type `α`. For example, `IO (Option α)` represents `IO` actions that don't always return a value of type `α`. This suggests the definition of the monad transformer `OptionT`:

```
def OptionT (m : Type u → Type v) (α : Type u) : Type v :=
  m (Option α)
```

As an example of `OptionT` in action, consider a program that asks the user questions. The function `getSomeInput` asks for a line of input and removes whitespace from both ends. If the resulting trimmed input is non-empty, then it is returned, but the function fails if there are no non-whitespace characters:

```
def getSomeInput : OptionT IO String := do
  let input ← (← IO.getStdin).getLine
  let trimmed := input.trim
  if trimmed == "" then
    failure
  else pure trimmed
```

This particular application tracks users with their name and their favorite species of beetle:

```
structure UserInfo where
  name : String
  favoriteBeetle : String
```

Asking the user for input is no more verbose than a function that uses only `IO` would be:

```
def getUserInfo : OptionT IO UserInfo := do
  IO.println "What is your name?"
  let name ← getSomeInput
  IO.println "What is your favorite species of beetle?"
  let beetle ← getSomeInput
  pure ⟨name, beetle⟩
```

However, because the function runs in an `OptionT IO` context rather than just in `IO`, failure in the first call to `getSomeInput` causes the whole `getUserInfo` to fail, with control never reaching the question about beetles. The main function, `interact`, invokes `getUserInfo` in a purely `IO` context, which allows it to check whether the call succeeded or failed by matching on the inner `Option`:

```
def interact : IO Unit := do
  match ← getUserInfo with
  | none =>
    IO.eprintln "Missing info"
  | some ⟨name, beetle⟩ =>
    IO.println s!"Hello {name}, whose favorite beetle is {beetle}."
```

The Monad Instance

Writing the monad instance reveals a difficulty. Based on the types, `pure` should use `pure` from the underlying monad `m` together with `some`. Just as `bind` for `Option` branches on the first argument, propagating `none`, `bind` for `OptionT` should run the monadic action that makes up the first argument, branch on the result, and then propagate `none`. Following this sketch yields the following definition, which Lean does not accept:

```
instance [Monad m] : Monad (OptionT m) where
  pure x := pure (some x)
  bind action next := do
    match (← action) with
    | none => pure none
    | some v => next v
```

The error message shows a cryptic type mismatch:

```
application type mismatch
  pure (some x)
argument
  some x
has type
  Option α✝ : Type ?u.25
but is expected to have type
  α✝ : Type ?u.25
```

The problem here is that Lean is selecting the wrong `Monad` instance for the surrounding use of `pure`. Similar errors occur for the definition of `bind`. One solution is to use type annotations to guide Lean to the correct `Monad` instance:

```
instance [Monad m] : Monad (OptionT m) where
  pure x := (pure (some x) : m (Option _))
  bind action next := (do
    match (← action) with
    | none => pure none
    | some v => next v : m (Option _))
```

While this solution works, it is inelegant and the code becomes a bit noisy.

An alternative solution is to define functions whose type signatures guide Lean to the correct instances. In fact, `OptionT` could have been defined as a structure:

```
structure OptionT (m : Type u → Type v) (α : Type u) : Type v where
  run : m (Option α)
```

This would solve the problem, because the constructor `OptionT.mk` and the field accessor `OptionT.run` would guide type class inference to the correct instances. The downside to doing this is that the resulting code is more complicated, and these structures can make it more difficult to read proofs. The best of both worlds can be achieved by defining functions that serve the same role as `OptionT.mk` and `OptionT.run`, but that work with the direct definition:

```
def OptionT.mk (x : m (Option α)) : OptionT m α := x

def OptionT.run (x : OptionT m α) : m (Option α) := x
```

Both functions return their inputs unchanged, but they indicate the boundary between code that is intended to present the interface of `OptionT` and code that is intended to present the interface of the underlying monad `m`. Using these helpers, the `Monad` instance becomes more readable:

```
instance [Monad m] : Monad (OptionT m) where
  pure x := OptionT.mk (pure (some x))
  bind action next := OptionT.mk do
    match ← action with
    | none => pure none
    | some v => next v
```

Here, the use of `OptionT.mk` indicates that its arguments should be considered as code that uses the interface of `m`, which allows Lean to select the correct `Monad` instances.

After defining the monad instance, it's a good idea to check that the monad contract is satisfied. The first step is to show that `bind (pure v) f` is the same as `f v`. Here's the steps:

```
bind (pure v) f
```

={ *Unfolding the definitions of* `bind` *and* `pure` }=

```
OptionT.mk do
  match ← pure (some v) with
  | none => pure none
  | some x => f x
```

={ *Desugaring nested action syntax* }=

```
OptionT.mk do
  let y ← pure (some v)
  match y with
  | none => pure none
  | some x => f x
```

={ *Desugaring* `do`*-notation* }=

```
OptionT.mk
  (pure (some v) >>= fun y =>
    match y with
    | none => pure none
    | some x => f x)
```

={ *Using the first monad rule for* `m` }=

```
        OptionT.mk
          (match some v with
           | none => pure none
           | some x => f x)
```

={ *Reduce* `match` }=

```
        OptionT.mk (f v)
```

={ *Definition of* `OptionT.mk` }=

```
        f v
```

The second rule states that `bind w pure` is the same as `w`. To demonstrate this, unfold the definitions of `bind` and `pure`, yielding:

```
OptionT.mk do
    match ← w with
    | none => pure none
    | some v => pure (some v)
```

In this pattern match, the result of both cases is the same as the pattern being matched, just with `pure` around it. In other words, it is equivalent to `w >>= fun y => pure y`, which is an instance of `m`'s second monad rule.

The final rule states that `bind (bind v f) g` is the same as `bind v (fun x => bind (f x) g)`. It can be checked in the same way, by expanding the definitions of `bind` and `pure` and then delegating to the underlying monad `m`.

An `Alternative` Instance

One convenient way to use `OptionT` is through the `Alternative` type class. Successful return is already indicated by `pure`, and the `failure` and `orElse` methods of `Alternative` provide a way to write a program that returns the first successful result from a number of subprograms:

```
instance [Monad m] : Alternative (OptionT m) where
  failure := OptionT.mk (pure none)
  orElse x y := OptionT.mk do
    match ← x with
    | some result => pure (some result)
    | none => y ()
```

Lifting

Lifting an action from `m` to `OptionT m` only requires wrapping `some` around the result of the computation:

```
instance [Monad m] : MonadLift m (OptionT m) where
  monadLift action := OptionT.mk do
    pure (some (← action))
```

Exceptions

The monad transformer version of `Except` is very similar to the monad transformer version of `Option`. Adding exceptions of type `ε` to some monadic action of type `m α` can be accomplished by adding exceptions to `α`, yielding type `m (Except ε α)`:

```
def ExceptT (ε : Type u) (m : Type u → Type v) (α : Type u) : Type v :=
  m (Except ε α)
```

`OptionT` provides `mk` and `run` functions to guide the type checker towards the correct `Monad` instances. This trick is also useful for `ExceptT`:

```
def ExceptT.mk {ε α : Type u} (x : m (Except ε α)) : ExceptT ε m α := x

def ExceptT.run {ε α : Type u} (x : ExceptT ε m α) : m (Except ε α) := x
```

The `Monad` instance for `ExceptT` is also very similar to the instance for `OptionT`. The only difference is that it propagates a specific error value, rather than `none`:

```
instance {ε : Type u} {m : Type u → Type v} [Monad m] : Monad (ExceptT ε m)
where
  pure x := ExceptT.mk (pure (Except.ok x))
  bind result next := ExceptT.mk do
    match ← result with
    | .error e => pure (.error e)
    | .ok x => next x
```

The type signatures of `ExceptT.mk` and `ExceptT.run` contain a subtle detail: they annotate the universe levels of `α` and `ε` explicitly. If they are not explicitly annotated, then Lean generates a more general type signature in which they have distinct polymorphic universe variables. However, the definition of `ExceptT` expects them to be in the same universe, because they can both be provided as arguments to `m`. This can lead to a problem in the `Monad` instance where the universe level solver fails to find a working solution:

```
def ExceptT.mk (x : m (Except ε α)) : ExceptT ε m α := x
```

```
 instance {ε : Type u} {m : Type u → Type v} [Monad m] : Monad (ExceptT ε m)
where
    pure x := ExceptT.mk (pure (Except.ok x))
    bind result next := ExceptT.mk do
      match (← result) with
      | .error e => pure (.error e)
      | .ok x => next x
message
"stuck at solving universe constraint
  max ?u.8974 ?u.8975 =?= u
while trying to unify
  ExceptT ε m α† : Type v
with
  ExceptT.{max ?u.8975 ?u.8974, v} ε m α† : Type v
```

This kind of error message is typically caused by underconstrained universe variables. Diagnosing it can be tricky, but a good first step is to look for reused universe variables in some definitions that are not reused in others.

Unlike `Option`, the `Except` datatype is typically not used as a data structure. It is always used as a control structure with its `Monad` instance. This means that it is reasonable to lift `Except ε` actions into `ExceptT ε m`, as well as actions from the underlying monad `m`. Lifting `Except` actions into `ExceptT` actions is done by wrapping them in `m`'s `pure`, because an action that only has exception effects cannot have any effects from the monad `m`:

```
instance [Monad m] : MonadLift (Except ε) (ExceptT ε m) where
  monadLift action := ExceptT.mk (pure action)
```

Because actions from `m` do not have any exceptions in them, their value should be wrapped in `Except.ok`. This can be accomplished using the fact that `Functor` is a superclass of `Monad`, so applying a function to the result of any monadic computation can be accomplished using `Functor.map`:

```
instance [Monad m] : MonadLift m (ExceptT ε m) where
  monadLift action := ExceptT.mk (.ok <$> action)
```

Type Classes for Exceptions

Exception handling fundamentally consists of two operations: the ability to throw exceptions, and the ability to recover from them. Thus far, this has been accomplished using the constructors of `Except` and pattern matching, respectively. However, this ties a program that uses exceptions to one specific encoding of the exception handling effect. Using a type class to capture these operations allows a program that uses exceptions to be used in *any* monad that supports throwing and catching.

Throwing an exception should take an exception as an argument, and it should be allowed in any context where a monadic action is requested. The "any context" part of the specification can be written as a type by writing `m α`—because there's no way to produce a value of any arbitrary type, the `throw` operation must be doing something that causes control to leave that part of the program. Catching an exception should accept any monadic action together with a handler, and the handler should explain how to get back to the action's type from an exception:

```
class MonadExcept (ε : outParam (Type u)) (m : Type v → Type w) where
  throw : ε → m α
  tryCatch : m α → (ε → m α) → m α
```

The universe levels on `MonadExcept` differ from those of `ExceptT`. In `ExceptT`, both `ε` and `α` have the same level, while `MonadExcept` imposes no such limitation. This is because `MonadExcept` never places an exception value inside of `m`. The most general universe signature recognizes the fact that `ε` and `α` are completely independent in this definition. Being more general means that the type class can be instantiated for a wider variety of types.

An example program that uses `MonadExcept` is a simple division service. The program is divided into two parts: a frontend that supplies a user interface based on strings that handles errors, and a backend that actually does the division. Both the frontend and the backend can throw exceptions, the former for ill-formed input and the latter for division by zero errors. The exceptions are an inductive type:

```
inductive Err where
  | divByZero
  | notANumber : String → Err
```

The backend checks for zero, and divides if it can:

```
def divBackend [Monad m] [MonadExcept Err m] (n k : Int) : m Int :=
  if k == 0 then
    throw .divByZero
  else pure (n / k)
```

The frontend's helper `asNumber` throws an exception if the string it is passed is not a number. The overall frontend converts its inputs to `Int` s and calls the backend, handling exceptions by returning a friendly string error:

```
def asNumber [Monad m] [MonadExcept Err m] (s : String) : m Int :=
  match s.toInt? with
  | none => throw (.notANumber s)
  | some i => pure i

def divFrontend [Monad m] [MonadExcept Err m] (n k : String) : m String :=
  tryCatch (do pure (toString (← divBackend (← asNumber n) (← asNumber k))))
    fun
      | .divByZero => pure "Division by zero!"
      | .notANumber s => pure s!"Not a number: \"{s}\""
```

Throwing and catching exceptions is common enough that Lean provides a special syntax for using `MonadExcept`. Just as `+` is short for `HAdd.hAdd`, `try` and `catch` can be used as shorthand for the `tryCatch` method:

```
def divFrontend [Monad m] [MonadExcept Err m] (n k : String) : m String :=
  try
    pure (toString (← divBackend (← asNumber n) (← asNumber k)))
  catch
    | .divByZero => pure "Division by zero!"
    | .notANumber s => pure s!"Not a number: \"{s}\""
```

In addition to `Except` and `ExceptT`, there are useful `MonadExcept` instances for other types that may not seem like exceptions at first glance. For example, failure due to `Option` can be seen as throwing an exception that contains no data whatsoever, so there is an instance of `MonadExcept Unit Option` that allows `try ... catch ...` syntax to be used with `Option`.

State

A simulation of mutable state is added to a monad by having monadic actions accept a starting state as an argument and return a final state together with their result. The bind operator for a state monad provides the final state of one action as an argument to the next action, threading the state through the program. This pattern can also be expressed as a monad transformer:

```
def StateT (σ : Type u) (m : Type u → Type v) (α : Type u) : Type (max u v) :=
  σ → m (α × σ)
```

Once again, the monad instance is very similar to that for `State`. The only difference is that the input and output states are passed around and returned in the underlying monad, rather than with pure code:

```
instance [Monad m] : Monad (StateT σ m) where
  pure x := fun s => pure (x, s)
  bind result next := fun s => do
    let (v, s') ← result s
    next v s'
```

The corresponding type class has `get` and `set` methods. One downside of `get` and `set` is that it becomes too easy to `set` the wrong state when updating it. This is because retrieving the state, updating it, and saving the updated state is a natural way to write some programs. For example, the following program counts the number of diacritic-free English vowels and consonants in a string of letters:

```
structure LetterCounts where
  vowels : Nat
  consonants : Nat
deriving Repr

inductive Err where
  | notALetter : Char → Err
deriving Repr

def vowels :=
  let lowerVowels := "aeiuoy"
  lowerVowels ++ lowerVowels.map (·.toUpper)

def consonants :=
  let lowerConsonants := "bcdfghjklmnpqrstvwxz"
  lowerConsonants ++ lowerConsonants.map (·.toUpper )

def countLetters (str : String) : StateT LetterCounts (Except Err) Unit :=
  let rec loop (chars : List Char) := do
    match chars with
    | [] => pure ()
    | c :: cs =>
      let st ← get
      let st' ←
        if c.isAlpha then
          if vowels.contains c then
            pure {st with vowels := st.vowels + 1}
          else if consonants.contains c then
            pure {st with consonants := st.consonants + 1}
          else -- modified or non-English letter
            pure st
        else throw (.notALetter c)
      set st'
      loop cs
  loop str.toList
```

It would be very easy to write `set st` instead of `set st'`. In a large program, this kind of mistake can lead to difficult-to-diagnose bugs.

While using a nested action for the call to `get` would solve this problem, it can't solve all such problems. For example, a function might update a field on a structure based on the values of two other fields. This would require two separate nested-action calls to `get`. Because the Lean compiler contains optimizations that are only effective when there is a single reference to a value, duplicating the references to the state might lead to code that is significantly slower. Both the potential performance problem and the potential bug can be worked around by using `modify`, which transforms the state using a function:

```
def countLetters (str : String) : StateT LetterCounts (Except Err) Unit :=
  let rec loop (chars : List Char) := do
    match chars with
    | [] => pure ()
    | c :: cs =>
      if c.isAlpha then
        if vowels.contains c then
          modify fun st => {st with vowels := st.vowels + 1}
        else if consonants.contains c then
          modify fun st => {st with consonants := st.consonants + 1}
        else -- modified or non-English letter
          pure ()
      else throw (.notALetter c)
      loop cs
  loop str.toList
```

The type class contains a function akin to `modify` called `modifyGet`, which allows the function to both compute a return value and transform an old state in a single step. The function returns a pair in which the first element is the return value, and the second element is the new state; `modify` just adds the constructor of `Unit` to the pair used in `modifyGet`:

```
def modify [MonadState σ m] (f : σ → σ) : m Unit :=
  modifyGet fun s => ((), f s)
```

The definition of `MonadState` is as follows:

```
class MonadState (σ : outParam (Type u)) (m : Type u → Type v) : Type (max (u+1)
v) where
  get : m σ
  set : σ → m PUnit
  modifyGet : (σ → α × σ) → m α
```

`PUnit` is a version of the `Unit` type that is universe-polymorphic to allow it to be in `Type u` instead of `Type`. While it would be possible to provide a default implementation of `modifyGet` in terms of `get` and `set`, it would not admit the optimizations that make `modifyGet` useful in the first place, rendering the method useless.

`Of` Classes and `The` Functions

Thus far, each monad type class that takes extra information, like the type of exceptions for `MonadExcept` or the type of the state for `MonadState`, has this type of extra information as an output parameter. For simple programs, this is generally convenient, because a monad that combines one use each of `StateT`, `ReaderT`, and `ExceptT` has only a single state type, environment type, and exception type. As monads grow in complexity, however, they may involve multiple states or errors types. In this case, the use of an output parameter makes it impossible to target both states in the same `do`-block.

For these cases, there are additional type classes in which the extra information is not an output parameter. These versions of the type classes use the word `Of` in the name. For example, `MonadStateOf` is like `MonadState`, but without an `outParam` modifier.

Instead of an `outParam`, these classes use a `semiOutParam` for their respective state, environment, or exception types. Like an `outParam`, a `semiOutParam` is not required be known before Lean begins the process of searching for an instance. However, there is an important difference: `outParams` are ignored during the search for an instance, and as a result they are truly outputs. If an `outParam` is known prior to the search, then Lean merely checks that the result of the search is the same as what was known. On the other hand, a `semiOutParam` that is known prior to the start of the search can be used to narrow down candidates, just like an input parameter.

When a state monad's state type is an `outParam`, then each monad can have at most one type of state. This is convenient, because it improves type inference: the state type can be inferred in more circumstances. This is also inconvenient, because a monad built from multiple uses of `StateT` cannot provide a useful `MonadState` instance. Using `MonadStateOf`, however, causes Lean to take the state type into account when it is available to select which instance to use, so one monad may provide multiple types of state. The downside of this is that the resulting instance may not be the one that was intended when the state type has not been specified explicitly enough, which can lead to confusing error messages.

Similarly, there are versions of the type class methods that accept the type of the extra information as an *explicit*, rather than implicit, argument. For `MonadStateOf`, there are `getThe` with type

```
(σ : Type u) → {m : Type u → Type v} → [MonadStateOf σ m] → m σ
```

and `modifyThe` with type

```
(σ : Type u) → {m : Type u → Type v} → [MonadStateOf σ m] → (σ → σ) → m PUnit
```

There is no `setThe` because the type of the new state is enough to decide which surrounding state monad transformer to use.

In the Lean standard library, there are instances of the non- `Of` versions of the classes defined in terms of the instances of the versions with `Of`. In other words, implementing the `Of` version yields implementations of both. It's generally a good idea to implement the `Of` version, and then start writing programs using the non- `Of` versions of the class, transitioning to the `Of` version if the output parameter becomes inconvenient.

Transformers and `Id`

The identity monad `Id` is the monad that has no effects whatsoever, to be used in contexts that expect a monad for some reason but where none is actually necessary. Another use of `Id` is to serve as the bottom of a stack of monad transformers. For instance, `StateT σ Id` works just like `State σ`.

Exercises

Monad Contract

Using pencil and paper, check that the rules of the monad transformer contract are satisfied for each monad transformer in this section.

Logging Transformer

Define a monad transformer version of `WithLog`. Also define the corresponding type class `MonadWithLog`, and write a program that combines logging and exceptions.

Counting Files

Modify `doug`'s monad with `StateT` such that it counts the number of directories and files seen. At the end of execution, it should display a report like:

```
Viewed 38 files in 5 directories.
```

Ordering Monad Transformers

When composing a monad from a stack of monad transformers, it's important to be aware that the order in which the monad transformers are layered matters. Different orderings of the same set of transformers result in different monads.

This version of `countLetters` is just like the previous version, except it uses type classes to describe the set of available effects instead of providing a concrete monad:

```
def countLetters [Monad m] [MonadState LetterCounts m] [MonadExcept Err m]
    (str : String) : m Unit :=
  let rec loop (chars : List Char) := do
    match chars with
    | [] => pure ()
    | c :: cs =>
      if c.isAlpha then
        if vowels.contains c then
          modify fun st => {st with vowels := st.vowels + 1}
        else if consonants.contains c then
          modify fun st => {st with consonants := st.consonants + 1}
        else -- modified or non-English letter
          pure ()
      else throw (.notALetter c)
      loop cs
  loop str.toList
```

The state and exception monad transformers can be combined in two different orders, each resulting in a monad that has instances of both type classes:

```
abbrev M1 := StateT LetterCounts (ExceptT Err Id)
abbrev M2 := ExceptT Err (StateT LetterCounts Id)
```

When run on input for which the program does not throw an exception, both monads yield similar results:

```
#eval countLetters (m := M1) "hello" ⟨0, 0⟩
```

```
Except.ok ((), { vowels := 2, consonants := 3 })
```

```
#eval countLetters (m := M2) "hello" ⟨0, 0⟩
```

```
(Except.ok (), { vowels := 2, consonants := 3 })
```

However, there is a subtle difference between these return values. In the case of `M1`, the outermost constructor is `Except.ok`, and it contains a pair of the unit constructor with the final state. In the case of `M2`, the outermost constructor is the pair, which contains `Except.ok` applied only to the unit constructor. The final state is outside of `Except.ok`. In both cases, the program returns the counts of vowels and consonants.

On the other hand, only one monad yields a count of vowels and consonants when the string causes an exception to be thrown. Using `M1`, only an exception value is returned:

```
#eval countLetters (m := M1) "hello!" ⟨0, 0⟩
```

```
Except.error (StEx.Err.notALetter '!')
```

Using `M2`, the exception value is paired with the state as it was at the time that the exception was thrown:

```
#eval countLetters (m := M2) "hello!" ⟨0, 0⟩
```

```
(Except.error (StEx.Err.notALetter '!'), { vowels := 2, consonants := 3 })
```

It might be tempting to think that `M2` is superior to `M1` because it provides more information that might be useful when debugging. The same program might compute *different* answers in `M1` than it does in `M2`, and there's no principled reason to say that one of these answers is necessarily better than the other. This can be seen by adding a step to the program that handles exceptions:

```
def countWithFallback
    [Monad m] [MonadState LetterCounts m] [MonadExcept Err m]
    (str : String) : m Unit :=
  try
    countLetters str
  catch _ =>
    countLetters "Fallback"
```

This program always succeeds, but it might succeed with different results. If no exception is thrown, then the results are the same as `countLetters`:

```
#eval countWithFallback (m := M1) "hello" ⟨0, 0⟩
```

```
Except.ok ((), { vowels := 2, consonants := 3 })
```

```
#eval countWithFallback (m := M2) "hello" ⟨0, 0⟩
```

```
(Except.ok (), { vowels := 2, consonants := 3 })
```

However, if the exception is thrown and caught, then the final states are very different. With `M1`, the final state contains only the letter counts from `"Fallback"`:

```
#eval countWithFallback (m := M1) "hello!" ⟨0, 0⟩
```

```
Except.ok ((), { vowels := 2, consonants := 6 })
```

With `M2`, the final state contains letter counts from both `"hello"` and from `"Fallback"`, as one would expect in an imperative language:

```
#eval countWithFallback (m := M2) "hello!" ⟨0, 0⟩
```

```
(Except.ok (), { vowels := 4, consonants := 9 })
```

In `M1`, throwing an exception "rolls back" the state to where the exception was caught. In `M2`, modifications to the state persist across the throwing and catching of exceptions. This difference can be seen by unfolding the definitions of `M1` and `M2`. `M1 α` unfolds to `LetterCounts → Except Err (α × LetterCounts)`, and `M2 α` unfolds to `LetterCounts → Except Err α × LetterCounts`. That is to say, `M1 α` describes functions that take an initial letter count, returning either an error or an `α` paired with updated counts. When an exception is thrown in `M1`, there is no final state. `M2 α` describes functions that take an initial letter count and return a new letter count paired with either an error or an `α`. When an exception is thrown in `M2`, it is accompanied by a state.

Commuting Monads

In the jargon of functional programming, two monad transformers are said to *commute* if they can be re-ordered without the meaning of the program changing. The fact that the result of the program can differ when `StateT` and `ExceptT` are reordered means that state and exceptions do not commute. In general, monad transformers should not be expected to commute.

Even though not all monad transformers commute, some do. For example, two uses of `StateT` can be re-ordered. Expanding the definitions in `StateT σ (StateT σ' Id) α` yields the type `σ → σ' → ((α × σ) × σ')`, and `StateT σ' (StateT σ Id) α` yields `σ' → σ → ((α × σ') × σ)`. In other words, the differences between them are that they nest the `σ` and `σ'` types in different places in the return type, and they accept their arguments in a different order. Any client code will still need to provide the same inputs, and it will still receive the same outputs.

Most programming languages that have both mutable state and exceptions work like `M2`. In those languages, state that *should* be rolled back when an exception is thrown is difficult to express, and it usually needs to be simulated in a manner that looks much like the passing of explicit state values in `M1`. Monad transformers grant the freedom to choose an interpretation of effect ordering that works for the problem at hand, with both choices being equally easy to program with. However, they also require care to be taken in the choice of ordering of transformers. With great expressive power comes the responsibility to check that what's being expressed is what is intended, and the type signature of `countWithFallback` is probably more polymorphic than it should be.

Exercises

- Check that `ReaderT` and `StateT` commute by expanding their definitions and reasoning about the resulting types.
- Do `ReaderT` and `ExceptT` commute? Check your answer by expanding their definitions and reasoning about the resulting types.
- Construct a monad transformer `ManyT` based on the definition of `Many`, with a suitable `Alternative` instance. Check that it satisfies the `Monad` contract.
- Does `ManyT` commute with `StateT`? If so, check your answer by expanding definitions and reasoning about the resulting types. If not, write a program in `ManyT (StateT σ Id)` and a program in `StateT σ (ManyT Id)`. Each program should be one that makes more sense for the given ordering of monad transformers.

More do Features

Lean's `do`-notation provides a syntax for writing programs with monads that resembles imperative programming languages. In addition to providing a convenient syntax for programs with monads, `do`-notation provides syntax for using certain monad transformers.

Single-Branched `if`

When working in a monad, a common pattern is to carry out a side effect only if some condition is true. For instance, `countLetters` contains a check for vowels or consonants, and letters that are neither have no effect on the state. This is captured by having the `else` branch evaluate to `pure ()`, which has no effects:

```
def countLetters (str : String) : StateT LetterCounts (Except Err) Unit :=
  let rec loop (chars : List Char) := do
    match chars with
    | [] => pure ()
    | c :: cs =>
      if c.isAlpha then
        if vowels.contains c then
          modify fun st => {st with vowels := st.vowels + 1}
        else if consonants.contains c then
          modify fun st => {st with consonants := st.consonants + 1}
        else -- modified or non-English letter
          pure ()
      else throw (.notALetter c)
      loop cs
  loop str.toList
```

When an `if` is a statement in a `do`-block, rather than being an expression, then `else pure ()` can simply be omitted, and Lean inserts it automatically. The following definition of `countLetters` is completely equivalent:

```
def countLetters (str : String) : StateT LetterCounts (Except Err) Unit :=
  let rec loop (chars : List Char) := do
    match chars with
    | [] => pure ()
    | c :: cs =>
      if c.isAlpha then
        if vowels.contains c then
          modify fun st => {st with vowels := st.vowels + 1}
        else if consonants.contains c then
          modify fun st => {st with consonants := st.consonants + 1}
      else throw (.notALetter c)
      loop cs
  loop str.toList
```

A program that uses a state monad to count the entries in a list that satisfy some monadic check can be written as follows:

```
def count [Monad m] [MonadState Nat m] (p : α → m Bool) : List α → m Unit
  | [] => pure ()
  | x :: xs => do
    if ← p x then
      modify (· + 1)
    count p xs
```

Similarly, `if not E1 then STMT...` can instead be written `unless E1 do STMT...`. The converse of `count` that counts entries that don't satisfy the monadic check can be written by replacing `if` with `unless`:

```
def countNot [Monad m] [MonadState Nat m] (p : α → m Bool) : List α → m Unit
  | [] => pure ()
  | x :: xs => do
    unless ← p x do
      modify (· + 1)
    countNot p xs
```

Understanding single-branched `if` and `unless` does not require thinking about monad transformers. They simply replace the missing branch with `pure ()`. The remaining extensions in this section, however, require Lean to automatically rewrite the `do`-block to add a local transformer on top of the monad that the `do`-block is written in.

Early Return

The standard library contains a function `List.find?` that returns the first entry in a list that satisfies some check. A simple implementation that doesn't make use of the fact that

`Option` is a monad loops over the list using a recursive function, with an `if` to stop the loop when the desired entry is found:

```
def List.find? (p : α → Bool) : List α → Option α
  | [] => none
  | x :: xs =>
    if p x then
      some x
    else
      find? p xs
```

Imperative languages typically sport the `return` keyword that aborts the execution of a function, immediately returning some value to the caller. In Lean, this is available in `do`-notation, and `return` halts the execution of a `do`-block, with `return`'s argument being the value returned from the monad. In other words, `List.find?` could have been written like this:

```
def List.find? (p : α → Bool) : List α → Option α
  | [] => failure
  | x :: xs => do
    if p x then return x
    find? p xs
```

Early return in imperative languages is a bit like an exception that can only cause the current stack frame to be unwound. Both early return and exceptions terminate execution of a block of code, effectively replacing the surrounding code with the thrown value. Behind the scenes, early return in Lean is implemented using a version of `ExceptT`. Each `do`-block that uses early return is wrapped in an exception handler (in the sense of the function `tryCatch`). Early returns are translated to throwing the value as an exception, and the handlers catch the thrown value and return it immediately. In other words, the `do`-block's original return value type is also used as the exception type.

Making this more concrete, the helper function `runCatch` strips a layer of `ExceptT` from the top of a monad transformer stack when the exception type and return type are the same:

```
def runCatch [Monad m] (action : ExceptT α m α) : m α := do
  match ← action with
  | Except.ok x => pure x
  | Except.error x => pure x
```

The `do`-block in `List.find?` that uses early return is translated to a `do`-block that does not use early return by wrapping it in a use of `runCatch`, and replacing early returns with `throw`:

```
def List.find? (p : α → Bool) : List α → Option α
  | [] => failure
  | x :: xs =>
    runCatch do
      if p x then throw x else pure ()
      monadLift (find? p xs)
```

Another situation in which early return is useful is command-line applications that terminate early if the arguments or input are incorrect. Many programs begin with a section that validates arguments and inputs before proceeding to the main body of the program. The following version of the greeting program `hello-name` checks that no command-line arguments were provided:

```
def main (argv : List String) : IO UInt32 := do
  let stdin ← IO.getStdin
  let stdout ← IO.getStdout
  let stderr ← IO.getStderr

  unless argv == [] do
    stderr.putStrLn s!"Expected no arguments, but got {argv.length}"
    return 1

  stdout.putStrLn "How would you like to be addressed?"
  stdout.flush

  let name := (← stdin.getLine).trim
  if name == "" then
    stderr.putStrLn s!"No name provided"
    return 1

  stdout.putStrLn s!"Hello, {name}!"

  return 0
```

Running it with no arguments and typing the name `David` yields the same result as the previous version:

```
$ lean --run EarlyReturn.lean
How would you like to be addressed?
David
Hello, David!
```

Providing the name as a command-line argument instead of an answer causes an error:

```
$ lean --run EarlyReturn.lean David
Expected no arguments, but got 1
```

And providing no name causes the other error:

```
$ lean --run EarlyReturn.lean
How would you like to be addressed?

No name provided
```

The program that uses early return avoids needing to nest the control flow, as is done in this version that does not use early return:

```
def main (argv : List String) : IO UInt32 := do
  let stdin ← IO.getStdin
  let stdout ← IO.getStdout
  let stderr ← IO.getStderr

  if argv != [] then
    stderr.putStrLn s!"Expected no arguments, but got {argv.length}"
    pure 1
  else
    stdout.putStrLn "How would you like to be addressed?"
    stdout.flush

    let name := (← stdin.getLine).trim
    if name == "" then
      stderr.putStrLn s!"No name provided"
      pure 1
    else
      stdout.putStrLn s!"Hello, {name}!"
      pure 0
```

One important difference between early return in Lean and early return in imperative languages is that Lean's early return applies only to the current `do`-block. When the entire definition of a function is in the same `do` block, this difference doesn't matter. But if `do` occurs underneath some other structures, then the difference becomes apparent. For example, given the following definition of `greet`:

```
def greet (name : String) : String :=
  "Hello, " ++ Id.run do return name
```

the expression `greet "David"` evaluates to `"Hello, David"`, not just `"David"`.

Loops

Just as every program with mutable state can be rewritten to a program that passes the state as arguments, every loop can be rewritten as a recursive function. From one perspective, `List.find?` is most clear as a recursive function. After all, its definition mirrors the structure of the list: if the head passes the check, then it should be returned; otherwise look in the tail. When no more entries remain, the answer is `none`. From another perspective, `List.find?` is most clear as a loop. After all, the program consults the entries in order until a satisfactory one is found, at which point it terminates. If the loop terminates without having returned, the answer is `none`.

Looping with ForM

Lean includes a type class that describes looping over a container type in some monad. This class is called `ForM`:

```
class ForM (m : Type u → Type v) (γ : Type w₁) (α : outParam (Type w₂)) where
  forM [Monad m] : γ → (α → m PUnit) → m PUnit
```

This class is quite general. The parameter `m` is a monad with some desired effects, `γ` is the collection to be looped over, and `α` is the type of elements from the collection. Typically, `m` is allowed to be any monad, but it is possible to have a data structure that e.g. only supports looping in `IO`. The method `forM` takes a collection, a monadic action to be run for its effects on each element from the collection, and is then responsible for running the actions.

The instance for `List` allows `m` to be any monad, it sets `γ` to be `List α`, and sets the class's `α` to be the same `α` found in the list:

```
def List.forM [Monad m] : List α → (α → m PUnit) → m PUnit
  | [], _ => pure ()
  | x :: xs, action => do
    action x
    forM xs action

instance : ForM m (List α) α where
  forM := List.forM
```

The function `doList` from `doug` is `forM` for lists. Because `forM` is intended to be used in `do`-blocks, it uses `Monad` rather than `Applicative`. `forM` can be used to make `countLetters` much shorter:

```
def countLetters (str : String) : StateT LetterCounts (Except Err) Unit :=
  forM str.toList fun c => do
    if c.isAlpha then
      if vowels.contains c then
        modify fun st => {st with vowels := st.vowels + 1}
      else if consonants.contains c then
        modify fun st => {st with consonants := st.consonants + 1}
    else throw (.notALetter c)
```

The instance for `Many` is very similar:

```
def Many.forM [Monad m] : Many α → (α → m PUnit) → m PUnit
  | Many.none, _ => pure ()
  | Many.more first rest, action => do
    action first
    forM (rest ()) action

instance : ForM m (Many α) α where
  forM := Many.forM
```

Because `γ` can be any type at all, `ForM` can support non-polymorphic collections. A very simple collection is one of the natural numbers less than some given number, in reverse order:

```
structure AllLessThan where
  num : Nat
```

Its `forM` operator applies the provided action to each smaller `Nat`:

```
def AllLessThan.forM [Monad m] (coll : AllLessThan) (action : Nat → m Unit) : m
Unit :=
  let rec countdown : Nat → m Unit
    | 0 => pure ()
    | n + 1 => do
      action n
      countdown n
  countdown coll.num

instance : ForM m AllLessThan Nat where
  forM := AllLessThan.forM
```

Running `IO.println` on each number less than five can be accomplished with `forM`:

```
#eval forM { num := 5 : AllLessThan } IO.println
```

```
4
3
2
1
0
```

An example `ForM` instance that works only in a particular monad is one that loops over the lines read from an IO stream, such as standard input:

```
structure LinesOf where
  stream : IO.FS.Stream

partial def LinesOf.forM (readFrom : LinesOf) (action : String → IO Unit) : IO
Unit := do
  let line ← readFrom.stream.getLine
  if line == "" then return ()
  action line
  forM readFrom action

instance : ForM IO LinesOf String where
  forM := LinesOf.forM
```

The definition of `forM` is marked `partial` because there is no guarantee that the stream is finite. In this case, `IO.FS.Stream.getLine` works only in the `IO` monad, so no other monad can be used for looping.

This example program uses this looping construct to filter out lines that don't contain letters:

```
def main (argv : List String) : IO UInt32 := do
  if argv != [] then
    IO.eprintln "Unexpected arguments"
    return 1

  forM (LinesOf.mk (← IO.getStdin)) fun line => do
    if line.any (·.isAlpha) then
      IO.print line

  return 0
```

The file `test-data` contains:

```
Hello!
!!!!!
12345
abc123

Ok
```

Invoking this program, which is stored in `ForMIO.lean`, yields the following output:

```
$ lean --run ForMIO.lean < test-data
Hello!
abc123
Ok
```

Stopping Iteration

Terminating a loop early is difficult to do with `forM`. Writing a function that iterates over the `Nat`s in an `AllLessThan` only until `3` is reached requires a means of stopping the loop partway through. One way to achieve this is to use `forM` with the `OptionT` monad transformer. The first step is to define `OptionT.exec`, which discards information about both the return value and whether or not the transformed computation succeeded:

```
def OptionT.exec [Applicative m] (action : OptionT m α) : m Unit :=
  action *> pure ()
```

Then, failure in the `OptionT` instance of `Alternative` can be used to terminate looping early:

```
def countToThree (n : Nat) : IO Unit :=
  let nums : AllLessThan := ⟨n⟩
  OptionT.exec (forM nums fun i => do
    if i < 3 then failure else IO.println i)
```

A quick test demonstrates that this solution works:

```
#eval countToThree 7
```

```
6
5
4
3
```

However, this code is not so easy to read. Terminating a loop early is a common task, and Lean provides more syntactic sugar to make this easier. This same function can also be written as follows:

```
def countToThree (n : Nat) : IO Unit := do
  let nums : AllLessThan := ⟨n⟩
  for i in nums do
    if i < 3 then break
    IO.println i
```

Testing it reveals that it works just like the prior version:

```
#eval countToThree 7
```

```
6
5
4
3
```

The `for ... in ... do ...` syntax desugars to the use of a type class called `ForIn`, which is a somewhat more complicated version of `ForM` that keeps track of state and early termination. The standard library provides an adapter that converts a `ForM` instance into a `ForIn` instance, called `ForM.forIn`. To enable `for` loops based on a `ForM` instance, add something like the following, with appropriate replacements for `AllLessThan` and `Nat`:

```
instance : ForIn m AllLessThan Nat where
  forIn := ForM.forIn
```

Note, however, that this adapter only works for `ForM` instances that keep the monad unconstrained, as most of them do. This is because the adapter uses `StateT` and `ExceptT`, rather than the underlying monad.

Early return is supported in `for` loops. The translation of `do` blocks with early return into a use of an exception monad transformer applies equally well underneath `forM` as the earlier use of `OptionT` to halt iteration does. This version of `List.find?` makes use of both:

```
def List.find? (p : α → Bool) (xs : List α) : Option α := do
  for x in xs do
    if p x then return x
  failure
```

In addition to `break`, `for` loops support `continue` to skip the rest of the loop body in an iteration. An alternative (but confusing) formulation of `List.find?` skips elements that don't satisfy the check:

```
def List.find? (p : α → Bool) (xs : List α) : Option α := do
  for x in xs do
    if not (p x) then continue
    return x
  failure
```

A `Range` is a structure that consists of a starting number, an ending number, and a step. They represent a sequence of natural numbers, from the starting number to the ending number, increasing by the step each time. Lean has special syntax to construct ranges, consisting of square brackets, numbers, and colons that comes in four varieties. The stopping point must always be provided, while the start and the step are optional, defaulting to `0` and `1`, respectively:

Expression	Start	Stop	Step	As List
`[:10]`	`0`	`10`	`1`	`[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]`
`[2:10]`	`2`	`10`	`1`	`[2, 3, 4, 5, 6, 7, 8, 9]`
`[:10:3]`	`0`	`10`	`3`	`[0, 3, 6, 9]`
`[2:10:3]`	`2`	`10`	`3`	`[2, 5, 8]`

Note that the starting number *is* included in the range, while the stopping numbers is not. All three arguments are `Nat` s, which means that ranges cannot count down—a range where the starting number is greater than or equal to the stopping number simply contains no numbers.

Ranges can be used with `for` loops to draw numbers from the range. This program counts even numbers from four to eight:

```
def fourToEight : IO Unit := do
  for i in [4:9:2] do
    IO.println i
```

Running it yields:

```
4
6
8
```

Finally, `for` loops support iterating over multiple collections in parallel, by separating the `in` clauses with commas. Looping halts when the first collection runs out of elements, so the declaration:

```
def parallelLoop := do
  for x in ["currant", "gooseberry", "rowan"], y in [4:8] do
    IO.println (x, y)
```

produces three lines of output:

```
#eval parallelLoop
```

```
(currant, 4)
(gooseberry, 5)
(rowan, 6)
```

Many data structures implement an enhanced version of the `ForIn` type class that adds evidence that the element was drawn from the collection to the loop body. These can be used by providing a name for the evidence prior to the name of the element. This function prints all the elements of an array together with their indices, and the compiler is able to determine that the array lookups are all safe due to the evidence `h`:

```
def printArray [ToString α] (xs : Array α) : IO Unit := do
  for h : i in [0:xs.size] do
    IO.println s!"{i}:\t{xs[i]}"
```

In this example, `h` is evidence that `i ∈ [0:xs.size]`, and the tactic that checks whether `xs[i]` is safe is able to transform this into evidence that `i < xs.size`.

Mutable Variables

In addition to early `return`, `else`-less `if`, and `for` loops, Lean supports local mutable variables within a `do` block. Behind the scenes, these mutable variables desugar to code that's equivalent to `StateT`, rather than being implemented by true mutable variables. Once again, functional programming is used to simulate imperative programming.

A local mutable variable is introduced with `let mut` instead of plain `let`. The definition `two`, which uses the identity monad `Id` to enable `do`-syntax without introducing any effects, counts to `2`:

```
def two : Nat := Id.run do
  let mut x := 0
  x := x + 1
  x := x + 1
  return x
```

This code is equivalent to a definition that uses `StateT` to add `1` twice:

```
def two : Nat :=
  let block : StateT Nat Id Nat := do
    modify (· + 1)
    modify (· + 1)
    return (← get)
  let (result, _finalState) := block 0
  result
```

Local mutable variables work well with all the other features of `do`-notation that provide convenient syntax for monad transformers. The definition `three` counts the number of entries in a three-entry list:

```
def three : Nat := Id.run do
  let mut x := 0
  for _ in [1, 2, 3] do
    x := x + 1
  return x
```

Similarly, `six` adds the entries in a list:

```
def six : Nat := Id.run do
  let mut x := 0
  for y in [1, 2, 3] do
    x := x + y
  return x
```

`List.count` counts the number of entries in a list that satisfy some check:

```
def List.count (p : α → Bool) (xs : List α) : Nat := Id.run do
  let mut found := 0
  for x in xs do
    if p x then found := found + 1
  return found
```

Local mutable variables can be more convenient to use and easier to read than an explicit local use of `StateT`. However, they don't have the full power of unrestricted mutable variables from imperative languages. In particular, they can only be modified in the `do`-block in which they are introduced. This means, for instance, that `for`-loops can't be replaced by otherwise-equivalent recursive helper functions. This version of `List.count`:

```
def List.count (p : α → Bool) (xs : List α) : Nat := Id.run do
  let mut found := 0
  let rec go : List α → Id Unit
    | [] => pure ()
    | y :: ys => do
      if p y then found := found + 1
      go ys
  return found
```

yields the following error on the attempted mutation of `found`:

```
`found` cannot be mutated, only variables declared using `let mut` can be
mutated. If you did not intend to mutate but define `found`, consider using `let
found` instead
```

This is because the recursive function is written in the identity monad, and only the monad of the `do`-block in which the variable is introduced is transformed with `StateT`.

What counts as a `do` block?

Many features of `do`-notation apply only to a single `do`-block. Early return terminates the current block, and mutable variables can only be mutated in the block that they are defined in. To use them effectively, it's important to know what counts as "the same block".

Generally speaking, the indented block following the `do` keyword counts as a block, and the immediate sequence of statements underneath it are part of that block. Statements in independent blocks that are nonetheless contained in a block are not considered part of the block. However, the rules that govern what exactly counts as the same block are slightly subtle, so some examples are in order. The precise nature of the rules can be tested by setting up a program with a mutable variable and seeing where the mutation is allowed. This program has a mutation that is clearly in the same block as the mutable variable:

```
example : Id Unit := do
  let mut x := 0
  x := x + 1
```

When a mutation occurs in a `do`-block that is part of a `let`-statement that defines a name using `:=`, then it is not considered to be part of the block:

```
example : Id Unit := do
  let mut x := 0
  let other := do
    x := x + 1
  other
```

```
`x` cannot be mutated, only variables declared using `let mut` can be mutated.
If you did not intend to mutate but define `x`, consider using `let x` instead
```

However, a `do` -block that occurs under a `let` -statement that defines a name using `←` is considered part of the surrounding block. The following program is accepted:

```
example : Id Unit := do
  let mut x := 0
  let other ← do
    x := x + 1
  pure other
```

Similarly, `do` -blocks that occur as arguments to functions are independent of their surrounding blocks. The following program is not accepted:

```
example : Id Unit := do
  let mut x := 0
  let addFour (y : Id Nat) := Id.run y + 4
  addFour do
    x := 5
```

```
`x` cannot be mutated, only variables declared using `let mut` can be mutated.
If you did not intend to mutate but define `x`, consider using `let x` instead
```

If the `do` keyword is completely redundant, then it does not introduce a new block. This program is accepted, and is equivalent to the first one in this section:

```
example : Id Unit := do
  let mut x := 0
  do x := x + 1
```

The contents of branches under a `do` (such as those introduced by `match` or `if`) are considered to be part of the surrounding block, whether or not a redundant `do` is added. The following programs are all accepted:

```
example : Id Unit := do
  let mut x := 0
  if x > 2 then
    x := x + 1

example : Id Unit := do
  let mut x := 0
  if x > 2 then do
    x := x + 1

example : Id Unit := do
  let mut x := 0
  match true with
  | true => x := x + 1
  | false => x := 17

example : Id Unit := do
  let mut x := 0
  match true with
  | true => do
    x := x + 1
  | false => do
    x := 17
```

Similarly, the `do` that occurs as part of the `for` and `unless` syntax is just part of their syntax, and does not introduce a fresh `do`-block. These programs are also accepted:

```
example : Id Unit := do
  let mut x := 0
  for y in [1:5] do
   x := x + y

example : Id Unit := do
  let mut x := 0
  unless 1 < 5 do
    x := x + 1
```

Imperative or Functional Programming?

The imperative features provided by Lean's `do`-notation allow many programs to very closely resemble their counterparts in languages like Rust, Java, or C#. This resemblance is very convenient when translating an imperative algorithm into Lean, and some tasks are just most naturally thought of imperatively. The introduction of monads and monad transformers enables imperative programs to be written in purely functional languages, and `do`-notation as a specialized syntax for monads (potentially locally transformed) allows functional programmers to have the best of both worlds: the strong reasoning principles afforded by immutability and a tight control over available effects through the type system are combined with syntax and libraries that allow programs that use effects to look familiar and be easy to read. Monads and monad transformers allow functional versus imperative programming to be a matter of perspective.

Exercises

- Rewrite `doug` to use `for` instead of the `doList` function. Are there other opportunities to use the features introduced in this section to improve the code? If so, use them!

Additional Conveniences

Pipe Operators

Functions are normally written before their arguments. When reading a program from left to right, this promotes a view in which the function's *output* is paramount—the function has a goal to achieve (that is, a value to compute), and it receives arguments to support it in this process. But some programs are easier to understand in terms of an input that is successively refined to produce the output. For these situations, Lean provides a *pipeline* operator which is similar to the that provided by F#. Pipeline operators are useful in the same situations as Clojure's threading macros.

The pipeline `E1 |> E2` is short for `E2 E1`. For example, evaluating:

```
#eval some 5 |> toString
```

results in:

```
"(some 5)"
```

While this change of emphasis can make some programs more convenient to read, pipelines really come into their own when they contain many components.

With the definition:

```
def times3 (n : Nat) : Nat := n * 3
```

the following pipeline:

```
#eval 5 |> times3 |> toString |> ("It is " ++ ·)
```

yields:

```
"It is 15"
```

More generally, a series of pipelines `E1 |> E2 |> E3 |> E4` is short for nested function applications `E4 (E3 (E2 E1))`.

Pipelines may also be written in reverse. In this case, they do not place the subject of data transformation first; however, in cases where many nested parentheses pose a challenge for readers, they can clarify the steps of application. The prior example could be equivalently written as:

```
#eval ("It is " ++ ·) <| toString <| times3 <| 5
```

which is short for:

```
#eval ("It is " ++ ·) (toString (times3 5))
```

Lean's method dot notation that uses the name of the type before the dot to resolve the namespace of the operator after the dot serves a similar purpose to pipelines. Even without the pipeline operator, it is possible to write `[1, 2, 3].reverse` instead of `List.reverse [1, 2, 3]`. However, the pipeline operator is also useful for dotted functions when using many of them. `([1, 2, 3].reverse.drop 1).reverse` can also be written as `[1, 2, 3] |> List.reverse |> List.drop 1 |> List.reverse`. This version avoids having to parenthesize expressions simply because they accept arguments, and it recovers the convenience of a chain of method calls in languages like Kotlin or C#. However, it still requires the namespace to be provided by hand. As a final convenience, Lean provides the "pipeline dot" operator, which groups functions like the pipeline but uses the name of the type to resolve namespaces. With "pipeline dot", the example can be rewritten to `[1, 2, 3] |>.reverse |>.drop 1 |>.reverse`.

Infinite Loops

Within a `do`-block, the `repeat` keyword introduces an infinite loop. For example, a program that spams the string `"Spam!"` can use it:

```
def spam : IO Unit := do
  repeat IO.println "Spam!"
```

A `repeat` loop supports `break` and `continue`, just like `for` loops.

The `dump` function from the implementation of `feline` uses a recursive function to run forever:

```
partial def dump (stream : IO.FS.Stream) : IO Unit := do
  let buf ← stream.read bufsize
  if buf.isEmpty then
    pure ()
  else
    let stdout ← IO.getStdout
    stdout.write buf
    dump stream
```

This function can be greatly shortened using `repeat`:

```
def dump (stream : IO.FS.Stream) : IO Unit := do
  let stdout ← IO.getStdout
  repeat do
    let buf ← stream.read bufsize
    if buf.isEmpty then break
    stdout.write buf
```

Neither `spam` nor `dump` need to be declared as `partial` because they are not themselves infinitely recursive. Instead, `repeat` makes use of a type whose `ForM` instance is `partial`. Partiality does not "infect" calling functions.

While Loops

When programming with local mutability, `while` loops can be a convenient alternative to `repeat` with an `if`-guarded `break`:

```
def dump (stream : IO.FS.Stream) : IO Unit := do
  let stdout ← IO.getStdout
  let mut buf ← stream.read bufsize
  while not buf.isEmpty do
    stdout.write buf
    buf ← stream.read bufsize
```

Behind the scenes, `while` is just a simpler notation for `repeat`.

Summary

Combining Monads

When writing a monad from scratch, there are design patterns that tend to describe the ways that each effect is added to the monad. Reader effects are added by having the monad's type be a function from the reader's environment, state effects are added by including a function from the initial state to the value paired with the final state, failure or exceptions are added by including a sum type in the return type, and logging or other output is added by including a product type in the return type. Existing monads can be made part of the return type as well, allowing their effects to be included in the new monad.

These design patterns are made into a library of reusable software components by defining *monad transformers*, which add an effect to some base monad. Monad transformers take the simpler monad types as arguments, returning the enhanced monad types. At a minimum, a monad transformer should provide the following instances:

1. A `Monad` instance that assumes the inner type is already a monad
2. A `MonadLift` instance to translate an action from the inner monad to the transformed monad

Monad transformers may be implemented as polymorphic structures or inductive datatypes, but they are most often implemented as functions from the underlying monad type to the enhanced monad type.

Type Classes for Effects

A common design pattern is to implement a particular effect by defining a monad that has the effect, a monad transformer that adds it to another monad, and a type class that provides a generic interface to the effect. This allows programs to be written that merely specify which effects they need, so the caller can provide any monad that has the right effects.

Sometimes, auxiliary type information (e.g. the state's type in a monad that provides state, or the exception's type in a monad that provides exceptions) is an output parameter, and sometimes it is not. The output parameter is most useful for simple programs that use each kind of effect only once, but it risks having the type checker commit to a the wrong type too early when multiple instances of the same effect are used in a given program. Thus, both versions are typically provided, with the ordinary-parameter version of the type class having a name that ends in `-Of` .

Monad Transformers Don't Commute

It is important to note that changing the order of transformers in a monad can change the meaning of programs that use the monad. For instance, re-ordering `StateT` and `ExceptT` can result either in programs that lose state modifications when exceptions are thrown or programs that keep changes. While most imperative languages provide only the latter, the increased flexibility provided by monad transformers demands thought and attention to choose the correct variety for the task at hand.

`do`-Notation for Monad Transformers

Lean's `do` -blocks support early return, in which the block is terminated with some value, locally mutable variables, `for` -loops with `break` and `continue` , and single-branched `if` - statements. While this may seem to be introducing imperative features that would get in the way of using Lean to write proofs, it is in fact nothing more than a more convenient syntax for certain common uses of monad transformers. Behind the scenes, whatever monad the `do` -block is written in is transformed by appropriate uses of `ExceptT` and `StateT` to support these additional effects.

Programming with Dependent Types

In most statically-typed programming languages, there is a hermetic seal between the world of types and the world of programs. Types and programs have different grammars and they are used at different times. Types are typically used at compile time, to check that a program obeys certain invariants. Programs are used at run time, to actually perform computations. When the two interact, it is usually in the form of a type-case operator like an "instance-of" check or a casting operator that provides the type checker with information that was otherwise unavailable, to be verified at run time. In other words, the interaction consists of types being inserted into the world of programs, where they gain some limited run-time meaning.

Lean does not impose this strict separation. In Lean, programs may compute types and types may contain programs. Placing programs in types allows their full computational power to be used at compile time, and the ability to return types from functions makes types into first-class participants in the programming process.

Dependent types are types that contain non-type expressions. A common source of dependent types is a named argument to a function. For example, the function `natOrStringThree` returns either a natural number or a string, depending on which `Bool` it is passed:

```
def natOrStringThree (b : Bool) : if b then Nat else String :=
  match b with
  | true => (3 : Nat)
  | false => "three"
```

Further examples of dependent types include:

- The introductory section on polymorphism contains `posOrNegThree`, in which the function's return type depends on the value of the argument.
- The `OfNat` type class depends on the specific natural number literal being used.
- The `CheckedInput` structure used in the example of validators depends on the year in which validation occurred.
- Subtypes contain propositions that refer to particular values.
- Essentially all interesting propositions, including those that determine the validity of array indexing notation, are types that contain values and are thus dependent types.

Dependent types vastly increase the power of a type system. The flexibility of return types that branch on argument values enables programs to be written that cannot easily be given types in other type systems. At the same time, dependent types allow a type signature to restrict which values may be returned from a function, enabling strong invariants to be enforced at compile time.

However, programming with dependent types can be quite complex, and it requires a whole set of skills above and beyond functional programming. Expressive specifications can be complicated to fulfill, and there is a real risk of tying oneself in knots and being unable to complete the program. On the other hand, this process can lead to new understanding, which can be expressed in a refined type that can be fulfilled. While this chapter scratches the surface of dependently typed programming, it is a deep topic that deserves an entire book of its own.

Indexed Families

Polymorphic inductive types take type arguments. For instance, `List` takes an argument that determines the type of the entries in the list, and `Except` takes arguments that determine the types of the exceptions or values. These type arguments, which are the same in every constructor of the datatype, are referred to as *parameters*.

Arguments to inductive types need not be the same in every constructor, however. Inductive types in which the arguments to the type vary based on the choice of constructor are called *indexed families*, and the arguments that vary are referred to as *indices*. The "hello world" of indexed families is a type of lists that contains the length of the list in addition to the type of entries, conventionally referred to as "vectors":

```
inductive Vect (α : Type u) : Nat → Type u where
   | nil : Vect α 0
   | cons : α → Vect α n → Vect α (n + 1)
```

The type of a vector of three `String`s includes the fact that it contains three `String`s:

```
example : Vect String 3 :=
  .cons "one" (.cons "two" (.cons "three" .nil))
```

Function declarations may take some arguments before the colon, indicating that they are available in the entire definition, and some arguments after, indicating a desire to pattern-match on them and define the function case by case. Inductive datatypes have a similar principle: the argument `α` is named at the top of the datatype declaration, prior to the colon, which indicates that it is a parameter that must be provided as the first argument in all occurrences of `Vect` in the definition, while the `Nat` argument occurs after the colon, indicating that it is an index that may vary. Indeed, the three occurrences of `Vect` in the `nil` and `cons` constructor declarations consistently provide `α` as the first argument, while the second argument is different in each case.

The declaration of `nil` states that it is a constructor of type `Vect α 0`. This means that using `Vect.nil` in a context expecting a `Vect String 3` is a type error, just as `[1, 2, 3]` is a type error in a context that expects a `List String`:

```
example : Vect String 3 := Vect.nil
```

```
type mismatch
  Vect.nil
has type
  Vect ?m.1605 0 : Type ?u.1604
but is expected to have type
  Vect String 3 : Type
```

The mismatch between `0` and `3` in this example plays exactly the same role as any other type mismatch, even though `0` and `3` are not themselves types. The metavariable in the message can be ignored because its presence indicates that `Vect.nil` can have any element type.

Indexed families are called *families* of types because different index values can make different constructors available for use. In some sense, an indexed family is not a type; rather, it is a collection of related types, and the choice of index values also chooses a type from the collection. Choosing the index `5` for `Vect` means that only the constructor `cons` is available, and choosing the index `0` means that only `nil` is available.

If the index is not yet known (e.g. because it is a variable), then no constructor can be used until it becomes known. Using `n` for the length allows neither `Vect.nil` nor `Vect.cons`, because there's no way to know whether the variable `n` should stand for a `Nat` that matches `0` or `n + 1`:

```
example : Vect String n := Vect.nil
```

```
type mismatch
  Vect.nil
has type
  Vect ?m.1602 0 : Type ?u.1576
but is expected to have type
  Vect String n : Type
```

```
example : Vect String n := Vect.cons "Hello" (Vect.cons "world" Vect.nil)
```

```
type mismatch
  Vect.cons "Hello" (Vect.cons "world" Vect.nil)
has type
  Vect String (0 + 1 + 1) : Type
but is expected to have type
  Vect String n : Type
```

Having the length of the list as part of its type means that the type becomes more informative. For example, `Vect.replicate` is a function that creates a `Vect` with a number of copies of a given value. The type that says this precisely is:

```
def Vect.replicate (n : Nat) (x : α) : Vect α n := _
```

The argument `n` appears as the length of the result. The message associated with the underscore placeholder describes the task at hand:

```
don't know how to synthesize placeholder
context:
α : Type u_1
n : Nat
x : α
⊢ Vect α n
```

When working with indexed families, constructors can only be applied when Lean can see that the constructor's index matches the index in the expected type. However, neither constructor has an index that matches `n` — `nil` matches `Nat.zero`, and `cons` matches `Nat.succ`. Just as in the example type errors, the variable `n` could stand for either, depending on which `Nat` is provided to the function as an argument. The solution is to use pattern matching to consider both of the possible cases:

```
def Vect.replicate (n : Nat) (x : α) : Vect α n :=
  match n with
  | 0 => _
  | k + 1 => _
```

Because `n` occurs in the expected type, pattern matching on `n` *refines* the expected type in the two cases of the match. In the first underscore, the expected type has become `Vect α 0`:

```
don't know how to synthesize placeholder
context:
α : Type u_1
n : Nat
x : α
⊢ Vect α 0
```

In the second underscore, it has become `Vect α (k + 1)`:

```
don't know how to synthesize placeholder
context:
α : Type u_1
n : Nat
x : α
k : Nat
⊢ Vect α (k + 1)
```

When pattern matching refines the type of a program in addition to discovering the structure of a value, it is called *dependent pattern matching*.

The refined type makes it possible to apply the constructors. The first underscore matches `Vect.nil`, and the second matches `Vect.cons`:

```
def Vect.replicate (n : Nat) (x : α) : Vect α n :=
  match n with
  | 0 => .nil
  | k + 1 => .cons _ _
```

The first underscore under the `.cons` should have type `α`. There is an `α` available, namely `x`:

```
don't know how to synthesize placeholder
context:
α : Type u_1
n : Nat
x : α
k : Nat
⊢ α
```

The second underscore should be a `Vect α k`, which can be produced by a recursive call to `replicate`:

```
don't know how to synthesize placeholder
context:
α : Type u_1
n : Nat
x : α
k : Nat
⊢ Vect α k
```

Here is the final definition of `replicate`:

```
def Vect.replicate (n : Nat) (x : α) : Vect α n :=
  match n with
  | 0 => .nil
  | k + 1 => .cons x (replicate k x)
```

In addition to providing assistance while writing the function, the informative type of `Vect.replicate` also allows client code to rule out a number of unexpected functions without having to read the source code. A version of `replicate` for lists could produce a list of the wrong length:

```
def List.replicate (n : Nat) (x : α) : List α :=
  match n with
  | 0 => []
  | k + 1 => x :: x :: replicate k x
```

However, making this mistake with `Vect.replicate` is a type error:

```
def Vect.replicate (n : Nat) (x : α) : Vect α n :=
  match n with
  | 0 => .nil
  | k + 1 => .cons x (.cons x (replicate k x))
```

```
application type mismatch
  cons x (cons x (replicate k x))
argument
  cons x (replicate k x)
has type
  Vect α (k + 1) : Type ?u.1578
but is expected to have type
  Vect α k : Type ?u.1578
```

The function `List.zip` combines two lists by pairing the first entry in the first list with the first entry in the second list, the second entry in the first list with the second entry in the second list, and so forth. `List.zip` can be used to pair the three highest peaks in the US state of Oregon with the three highest peaks in Denmark:

```
["Mount Hood",
 "Mount Jefferson",
 "South Sister"].zip ["Møllehøj", "Yding Skovhøj", "Ejer Bavnehøj"]
```

The result is a list of three pairs:

```
[("Mount Hood", "Møllehøj"),
 ("Mount Jefferson", "Yding Skovhøj"),
 ("South Sister", "Ejer Bavnehøj")]
```

It's somewhat unclear what should happen when the lists have different lengths. Like many languages, Lean chooses to ignore the extra entries in one of the lists. For instance, combining the heights of the five highest peaks in Oregon with those of the three highest peaks in Denmark yields three pairs. In particular,

```
[3428.8, 3201, 3158.5, 3075, 3064].zip [170.86, 170.77, 170.35]
```

evaluates to

```
[(3428.8, 170.86), (3201, 170.77), (3158.5, 170.35)]
```

While this approach is convenient because it always returns an answer, it runs the risk of throwing away data when the lists unintentionally have different lengths. F# takes a different approach: its version of `List.zip` throws an exception when the lengths don't match, as can be seen in this `fsi` session:

```
> List.zip [3428.8; 3201.0; 3158.5; 3075.0; 3064.0] [170.86; 170.77; 170.35];;
```

```
System.ArgumentException: The lists had different lengths.
list2 is 2 elements shorter than list1 (Parameter 'list2')
   at Microsoft.FSharp.Core.DetailedExceptions.invalidArgDifferentListLength[?]
(String arg1, String arg2, Int32 diff) in /builddir/build/BUILD/dotnet-v3.1.424-
SDK/src/fsharp.3ef6f0b514198c0bfa6c2c09fefe41a740b024d5/src/fsharp/FSharp.Core/
local.fs:line 24
   at Microsoft.FSharp.Primitives.Basics.List.zipToFreshConsTail[a,b]
(FSharpList`1 cons, FSharpList`1 xs1, FSharpList`1 xs2) in /builddir/build/
BUILD/dotnet-v3.1.424-SDK/src/fsharp.3ef6f0b514198c0bfa6c2c09fefe41a740b024d5/
src/fsharp/FSharp.Core/local.fs:line 918
   at Microsoft.FSharp.Primitives.Basics.List.zip[T1,T2](FSharpList`1 xs1,
FSharpList`1 xs2) in /builddir/build/BUILD/dotnet-v3.1.424-SDK/src/
fsharp.3ef6f0b514198c0bfa6c2c09fefe41a740b024d5/src/fsharp/FSharp.Core/
local.fs:line 929
   at Microsoft.FSharp.Collections.ListModule.Zip[T1,T2](FSharpList`1 list1,
FSharpList`1 list2) in /builddir/build/BUILD/dotnet-v3.1.424-SDK/src/
fsharp.3ef6f0b514198c0bfa6c2c09fefe41a740b024d5/src/fsharp/FSharp.Core/
list.fs:line 466
   at <StartupCode$FSI_0006>.$FSI_0006.main@()
Stopped due to error
```

This avoids accidentally discarding information, but crashing a program comes with its own difficulties. The Lean equivalent, which would use the `Option` or `Except` monads, would introduce a burden that may not be worth the safety.

Using `Vect`, however, it is possible to write a version of `zip` with a type that requires that both arguments have the same length:

```
def Vect.zip : Vect α n → Vect β n → Vect (α × β) n
  | .nil, .nil => .nil
  | .cons x xs, .cons y ys => .cons (x, y) (zip xs ys)
```

This definition only has patterns for the cases where either both arguments are `Vect.nil` or both arguments are `Vect.cons`, and Lean accepts the definition without a "missing cases" error like the one that results from a similar definition for `List`:

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

```
missing cases:
(List.cons _ _), []
[], (List.cons _ _)
```

This is because the constructor used in the first pattern, `nil` or `cons`, *refines* the type checker's knowledge about the length `n`. When the first pattern is `nil`, the type checker can additionally determine that the length was `0`, so the only possible choice for the second pattern is `nil`. Similarly, when the first pattern is `cons`, the type checker can determine that the length was `k+1` for some `Nat` `k`, so the only possible choice for the second pattern is `cons`. Indeed, adding a case that uses `nil` and `cons` together is a type error, because the lengths don't match:

```
def Vect.zip : Vect α n → Vect β n → Vect (α × β) n
  | .nil, .nil => .nil
  | .nil, .cons y ys => .nil
  | .cons x xs, .cons y ys => .cons (x, y) (zip xs ys)
```

```
type mismatch
  Vect.cons y ys
has type
  Vect ?m.3469 (?m.3480 + 1) : Type ?u.3477
but is expected to have type
  Vect β 0 : Type ?u.3344
```

The refinement of the length can be observed by making `n` into an explicit argument:

```
def Vect.zip : (n : Nat) → Vect α n → Vect β n → Vect (α × β) n
  | 0, .nil, .nil => .nil
  | k + 1, .cons x xs, .cons y ys => .cons (x, y) (zip k xs ys)
```

Exercises

Getting a feel for programming with dependent types requires experience, and the exercises in this section are very important. For each exercise, try to see which mistakes the type checker can catch, and which ones it can't, by experimenting with the code as you go. This is also a good way to develop a feel for the error messages.

- Double-check that `Vect.zip` gives the right answer when combining the three highest peaks in Oregon with the three highest peaks in Denmark. Because `Vect` doesn't have the syntactic sugar that `List` has, it can be helpful to begin by defining `oregonianPeaks : Vect String 3` and `danishPeaks : Vect String 3`.

- Define a function `Vect.map` with type `(α → β) → Vect α n → Vect β n`.
- Define a function `Vect.zipWith` that combines the entries in a `Vect` one at a time with a function. It should have the type `(α → β → γ) → Vect α n → Vect β n → Vect γ n`.
- Define a function `Vect.unzip` that splits a `Vect` of pairs into a pair of `Vect`s. It should have the type `Vect (α × β) n → Vect α n × Vect β n`.
- Define a function `Vect.push` that adds an entry to the *end* of a `Vect`. Its type should be `Vect α n → α → Vect α (n + 1)` and `#eval Vect.push (.cons "snowy" .nil) "peaks"` should yield `Vect.cons "snowy" (Vect.cons "peaks" (Vect.nil))`.
- Define a function `Vect.reverse` that reverses the order of a `Vect`.
- Define a function `Vect.drop` with the following type: `(n : Nat) → Vect α (k + n) → Vect α k`. Verify that it works by checking that `#eval danishPeaks.drop 2` yields `Vect.cons "Ejer Bavnehøj" (Vect.nil)`.
- Define a function `Vect.take` with type `(n : Nat) → Vect α (k + n) → Vect α n` that returns the first `n` entries in the `Vect`. Check that it works on an example.

The Universe Design Pattern

In Lean, types such as `Type`, `Type 3`, and `Prop` that classify other types are known as universes. However, the term *universe* is also used for a design pattern in which a datatype is used to represent a subset of Lean's types, and a function converts the datatype's constructors into actual types. The values of this datatype are called *codes* for their types.

Just like Lean's built-in universes, the universes implemented with this pattern are types that describe some collection of available types, even though the mechanism by which it is done is different. In Lean, there are types such as `Type`, `Type 3`, and `Prop` that directly describe other types. This arrangement is referred to as *universes à la Russell*. The user-defined universes described in this section represent all of their types as *data*, and include an explicit function to interpret these codes into actual honest-to-goodness types. This arrangement is referred to as *universes à la Tarski*. While languages such as Lean that are based on dependent type theory almost always use Russell-style universes, Tarski-style universes are a useful pattern for defining APIs in these languages.

Defining a custom universe makes it possible to carve out a closed collection of types that can be used with an API. Because the collection of types is closed, recursion over the codes allows programs to work for *any* type in the universe. One example of a custom universe has the codes `nat`, standing for `Nat`, and `bool`, standing for `Bool`:

```
inductive NatOrBool where
  | nat | bool

abbrev NatOrBool.asType (code : NatOrBool) : Type :=
  match code with
  | .nat => Nat
  | .bool => Bool
```

Pattern matching on a code allows the type to be refined, just as pattern matching on the constructors of `Vect` allows the expected length to be refined. For instance, a program that deserializes the types in this universe from a string can be written as follows:

```
def decode (t : NatOrBool) (input : String) : Option t.asType :=
  match t with
  | .nat => input.toNat?
  | .bool =>
    match input with
    | "true" => some true
    | "false" => some false
    | _ => none
```

Dependent pattern matching on `t` allows the expected result type `t.asType` to be respectively refined to `NatOrBool.nat.asType` and `NatOrBool.bool.asType`, and these compute to the actual types `Nat` and `Bool`.

Like any other data, codes may be recursive. The type `NestedPairs` codes for any possible nesting of the pair and natural number types:

```
inductive NestedPairs where
  | nat : NestedPairs
  | pair : NestedPairs → NestedPairs → NestedPairs

abbrev NestedPairs.asType : NestedPairs → Type
  | .nat => Nat
  | .pair t1 t2 => asType t1 × asType t2
```

In this case, the interpretation function `NestedPairs.asType` is recursive. This means that recursion over codes is required in order to implement `BEq` for the universe:

```
def NestedPairs.beq (t : NestedPairs) (x y : t.asType) : Bool :=
  match t with
  | .nat => x == y
  | .pair t1 t2 => beq t1 x.fst y.fst && beq t2 x.snd y.snd

instance {t : NestedPairs} : BEq t.asType where
  beq x y := t.beq x y
```

Even though every type in the `NestedPairs` universe already has a `BEq` instance, type class search does not automatically check every possible case of a datatype in an instance declaration, because there might be infinitely many such cases, as with `NestedPairs`. Attempting to appeal directly to the `BEq` instances rather than explaining to Lean how to find them by recursion on the codes results in an error:

```
instance {t : NestedPairs} : BEq t.asType where
  beq x y := x == y
```

```
failed to synthesize
  BEq t.asType

Additional diagnostic information may be available using the `set_option
diagnostics true` command.
```

The `t` in the error message stands for an unknown value of type `NestedPairs`.

Type Classes vs Universes

Type classes allow an open-ended collection of types to be used with an API as long as they have implementations of the necessary interfaces. In most cases, this is preferable. It is hard to predict all use cases for an API ahead of time, and type classes are a convenient way to allow library code to be used with more types than the original author expected.

A universe à la Tarski, on the other hand, restricts the API to be usable only with a predetermined collection of types. This is useful in a few situations:

- When a function should act very differently depending on which type it is passed—it is impossible to pattern match on types themselves, but pattern matching on codes for types is allowed
- When an external system inherently limits the types of data that may be provided, and extra flexibility is not desired
- When additional properties of a type are required over and above the implementation of some operations

Type classes are useful in many of the same situations as interfaces in Java or C#, while a universe à la Tarski can be useful in cases where a sealed class might be used, but where an ordinary inductive datatype is not usable.

A Universe of Finite Types

Restricting the types that can be used with an API to a predetermined collection can enable operations that would be impossible for an open-ended API. For example, functions can't normally be compared for equality. Functions should be considered equal when they map the same inputs to the same outputs. Checking this could take infinite amounts of time, because comparing two functions with type `Nat → Bool` would require checking that the functions returned the same `Bool` for each and every `Nat`.

In other words, a function from an infinite type is itself infinite. Functions can be viewed as tables, and a function whose argument type is infinite requires infinitely many rows to represent each case. But functions from finite types require only finitely many rows in their tables, making them finite. Two functions whose argument type is finite can be checked for equality by enumerating all possible arguments, calling the functions on each of them, and then comparing the results. Checking higher-order functions for equality requires generating all possible functions of a given type, which additionally requires that the return type is finite so that each element of the argument type can be mapped to each element of the return type. This is not a *fast* method, but it does complete in finite time.

One way to represent finite types is by a universe:

```
inductive Finite where
  | unit : Finite
  | bool : Finite
  | pair : Finite → Finite → Finite
  | arr : Finite → Finite → Finite

abbrev Finite.asType : Finite → Type
  | .unit => Unit
  | .bool => Bool
  | .pair t1 t2 => asType t1 × asType t2
  | .arr t1 t2 => asType t1 → asType t2
```

In this universe, the constructor `arr` stands for the function type, which is written with an `arr` ow.

Comparing two values from this universe for equality is almost the same as in the `NestedPairs` universe. The only important difference is the addition of the case for `arr`, which uses a helper called `Finite.enumerate` to generate every value from the type coded for by `t1`, checking that the two functions return equal results for every possible input:

```
def Finite.beq (t : Finite) (x y : t.asType) : Bool :=
  match t with
  | .unit => true
  | .bool => x == y
  | .pair t1 t2 => beq t1 x.fst y.fst && beq t2 x.snd y.snd
  | .arr t1 t2 =>
    t1.enumerate.all fun arg => beq t2 (x arg) (y arg)
```

The standard library function `List.all` checks that the provided function returns `true` on every entry of a list. This function can be used to compare functions on the Booleans for equality:

```
#eval Finite.beq (.arr .bool .bool) (fun _ => true) (fun b => b == b)
```

```
true
```

It can also be used to compare functions from the standard library:

```
#eval Finite.beq (.arr .bool .bool) (fun _ => true) not
```

```
false
```

It can even compare functions built using tools such as function composition:

```
#eval Finite.beq (.arr .bool .bool) id (not ∘ not)
```

```
true
```

This is because the `Finite` universe codes for Lean's *actual* function type, not a special analogue created by the library.

The implementation of `enumerate` is also by recursion on the codes from `Finite`.

```
def Finite.enumerate (t : Finite) : List t.asType :=
  match t with
  | .unit => [()]
  | .bool => [true, false]
  | .pair t1 t2 => t1.enumerate.product t2.enumerate
  | .arr t1 t2 => t1.functions t2.enumerate
```

In the case for `Unit`, there is only a single value. In the case for `Bool`, there are two values to return (`true` and `false`). In the case for pairs, the result should be the Cartesian product of the values for the type coded for by `t1` and the values for the type coded for by `t2`. In other words, every value from `t1` should be paired with every value from `t2`. The helper function `List.product` can certainly be written with an ordinary recursive function, but here it is defined using `for` in the identity monad:

```
def List.product (xs : List α) (ys : List β) : List (α × β) := Id.run do
  let mut out : List (α × β) := []
  for x in xs do
    for y in ys do
      out := (x, y) :: out
  pure out.reverse
```

Finally, the case of `Finite.enumerate` for functions delegates to a helper called `Finite.functions` that takes a list of all of the return values to target as an argument.

Generally speaking, generating all of the functions from some finite type to a collection of result values can be thought of as generating the functions' tables. Each function assigns an output to each input, which means that a given function has k rows in its table when there are k possible arguments. Because each row of the table could select any of n possible outputs, there are n^k potential functions to generate.

Once again, generating the functions from a finite type to some list of values is recursive on the code that describes the finite type:

```
def Finite.functions (t : Finite) (results : List α) : List (t.asType → α) :=
  match t with
```

The table for functions from `Unit` contains one row, because the function can't pick different results based on which input it is provided. This means that one function is generated for each potential input.

```
| .unit =>
  results.map fun r =>
    fun () => r
```

There are n^2 functions from `Bool` when there are n result values, because each individual function of type `Bool → α` uses the `Bool` to select between two particular `α` s:

```
| .bool =>
  (results.product results).map fun (r1, r2) =>
    fun
      | true => r1
      | false => r2
```

Generating the functions from pairs can be achieved by taking advantage of currying. A function from a pair can be transformed into a function that takes the first element of the pair and returns a function that's waiting for the second element of the pair. Doing this allows `Finite.functions` to be used recursively in this case:

```
| .pair t1 t2 =>
  let f1s := t1.functions <| t2.functions results
  f1s.map fun f =>
    fun (x, y) =>
      f x y
```

Generating higher-order functions is a bit of a brain bender. Each higher-order function takes a function as its argument. This argument function can be distinguished from other functions based on its input/output behavior. In general, the higher-order function can apply the argument function to every possible argument, and it can then carry out any possible behavior based on the result of applying the argument function. This suggests a means of constructing the higher-order functions:

- Begin with a list of all possible arguments to the function that is itself an argument.
- For each possible argument, construct all possible behaviors that can result from the observation of applying the argument function to the possible argument. This can be done using `Finite.functions` and recursion over the rest of the possible arguments, because the result of the recursion represents the functions based on the observations of the rest of the possible arguments. `Finite.functions` constructs all the ways of achieving these based on the observation for the current argument.
- For potential behavior in response to these observations, construct a higher-order function that applies the argument function to the current possible argument. The result of this is then passed to the observation behavior.

- The base case of the recursion is a higher-order function that observes nothing for each result value—it ignores the argument function and simply returns the result value.

Defining this recursive function directly causes Lean to be unable to prove that the whole function terminates. However, using a simpler form of recursion called a *right fold* can be used to make it clear to the termination checker that the function terminates. A right fold takes three arguments: a step function that combines the head of the list with the result of the recursion over the tail, a default value to return when the list is empty, and the list being processed. It then analyzes the list, essentially replacing each `::` in the list with a call to the step function and replacing `[]` with the default value:

```
def List.foldr (f : α → β → β) (default : β) : List α → β
  | []     => default
  | a :: l => f a (foldr f default l)
```

Finding the sum of the `Nat`s in a list can be done with `foldr`:

```
[1, 2, 3, 4, 5].foldr (· + ·) 0
===>
(1 :: 2 :: 3 :: 4 :: 5 :: []).foldr (· + ·) 0
===>
(1 + 2 + 3 + 4 + 5 + 0)
===>
15
```

With `foldr`, the higher-order functions can be created as follows:

```
    | .arr t1 t2 =>
      let args := t1.enumerate
      let base :=
        results.map fun r =>
          fun _ => r
      args.foldr
        (fun arg rest =>
          (t2.functions rest).map fun more =>
            fun f => more (f arg) f)
        base
```

The complete definition of `Finite.Functions` is:

```
  def Finite.functions (t : Finite) (results : List α) : List (t.asType → α) :=
    match t with
      | .unit =>
        results.map fun r =>
          fun () => r
      | .bool =>
        (results.product results).map fun (r1, r2) =>
          fun
            | true => r1
            | false => r2
      | .pair t1 t2 =>
        let f1s := t1.functions <| t2.functions results
        f1s.map fun f =>
          fun (x, y) =>
            f x y
      | .arr t1 t2 =>
        let args := t1.enumerate
        let base :=
          results.map fun r =>
            fun _ => r
        args.foldr
          (fun arg rest =>
            (t2.functions rest).map fun more =>
              fun f => more (f arg) f)
          base
```

Because `Finite.enumerate` and `Finite.functions` call each other, they must be defined in a `mutual` block. In other words, right before the definition of `Finite.enumerate` is the `mutual` keyword:

```
mutual
  def Finite.enumerate (t : Finite) : List t.asType :=
    match t with
```

and right after the definition of `Finite.functions` is the `end` keyword:

```
      | .arr t1 t2 =>
        let args := t1.enumerate
        let base :=
          results.map fun r =>
            fun _ => r
        args.foldr
          (fun arg rest =>
            (t2.functions rest).map fun more =>
              fun f => more (f arg) f)
          base
end
```

This algorithm for comparing functions is not particularly practical. The number of cases to check grows exponentially; even a simple type like `((Bool × Bool) → Bool) → Bool` describes 65536 distinct functions. Why are there so many? Based on the reasoning above, and using $|T|$ to represent the number of values described by the type T, we should expect that

$$|((\mathtt{Bool} \times \mathtt{Bool}) \rightarrow \mathtt{Bool}) \rightarrow \mathtt{Bool}|$$

is

$$|\mathrm{Bool}|^{|(\mathtt{Bool} \times \mathtt{Bool}) \rightarrow \mathtt{Bool}|},$$

which is

$$2^{2^{|\mathtt{Bool} \times \mathtt{Bool}|}},$$

which is

$$2^{2^4}$$

or 65536. Nested exponentials grow quickly, and there are many higher-order functions.

Exercises

- Write a function that converts any value from a type coded for by `Finite` into a string. Functions should be represented as their tables.
- Add the empty type `Empty` to `Finite` and `Finite.beq`.
- Add `Option` to `Finite` and `Finite.beq`.

Worked Example: Typed Queries

Indexed families are very useful when building an API that is supposed to resemble some other language. They can be used to write a library of HTML constructors that don't permit generating invalid HTML, to encode the specific rules of a configuration file format, or to model complicated business constraints. This section describes an encoding of a subset of relational algebra in Lean using indexed families, as a simpler demonstration of techniques that can be used to build a more powerful database query language.

This subset uses the type system to enforce requirements such as disjointness of field names, and it uses type-level computation to reflect the schema into the types of values that are returned from a query. It is not a realistic system, however—databases are represented as linked lists of linked lists, the type system is much simpler than that of SQL, and the operators of relational algebra don't really match those of SQL. However, it is large enough to demonstrate useful principles and techniques.

A Universe of Data

In this relational algebra, the base data that can be held in columns can have types `Int`, `String`, and `Bool` and are described by the universe `DBType`:

```
inductive DBType where
  | int | string | bool

abbrev DBType.asType : DBType → Type
  | .int => Int
  | .string => String
  | .bool => Bool
```

Using `asType` allows these codes to be used for types. For example:

```
#eval ("Mount Hood" : DBType.string.asType)
```

```
"Mount Hood"
```

It is possible to compare the values described by any of the three database types for equality. Explaining this to Lean, however, requires a bit of work. Simply using `BEq` directly fails:

```
def DBType.beq (t : DBType) (x y : t.asType) : Bool :=
  x == y
```

```
failed to synthesize
  BEq t.asType

Additional diagnostic information may be available using the `set_option
diagnostics true` command.
```

Just as in the nested pairs universe, type class search doesn't automatically check each possibility for `t`'s value The solution is to use pattern matching to refine the types of `x` and `y`:

```
def DBType.beq (t : DBType) (x y : t.asType) : Bool :=
  match t with
  | .int => x == y
  | .string => x == y
  | .bool => x == y
```

In this version of the function, `x` and `y` have types `Int`, `String`, and `Bool` in the three respective cases, and these types all have `BEq` instances. The definition of `dbEq` can be used to define a `BEq` instance for the types that are coded for by `DBType`:

```
instance {t : DBType} : BEq t.asType where
  beq := t.beq
```

This is not the same as an instance for the codes:

```
instance : BEq DBType where
  beq
    | .int, .int => true
    | .string, .string => true
    | .bool, .bool => true
    | _, _ => false
```

The former instance allows comparison of values drawn from the types described by the codes, while the latter allows comparison of the codes themselves.

A `Repr` instance can be written using the same technique. The method of the `Repr` class is called `reprPrec` because it is designed to take things like operator precedence into account when displaying values. Refining the type through dependent pattern matching allows the `reprPrec` methods from the `Repr` instances for `Int`, `String`, and `Bool` to be used:

```
instance {t : DBType} : Repr t.asType where
  reprPrec :=
    match t with
    | .int => reprPrec
    | .string => reprPrec
    | .bool => reprPrec
```

Schemas and Tables

A schema describes the name and type of each column in a database:

```
structure Column where
  name : String
  contains : DBType

abbrev Schema := List Column
```

In fact, a schema can be seen as a universe that describes rows in a table. The empty schema describes the unit type, a schema with a single column describes that value on its own, and a schema with at least two columns is represented by a tuple:

```
abbrev Row : Schema → Type
  | [] => Unit
  | [col] => col.contains.asType
  | col1 :: col2 :: cols => col1.contains.asType × Row (col2::cols)
```

As described in the initial section on product types, Lean's product type and tuples are right-associative. This means that nested pairs are equivalent to ordinary flat tuples.

A table is a list of rows that share a schema:

```
abbrev Table (s : Schema) := List (Row s)
```

For example, a diary of visits to mountain peaks can be represented with the schema `peak`:

```
abbrev peak : Schema := [
  ⟨"name", .string⟩,
  ⟨"location", .string⟩,
  ⟨"elevation", .int⟩,
  ⟨"lastVisited", .int⟩
]
```

A selection of peaks visited by the author of this book appears as an ordinary list of tuples:

```
def mountainDiary : Table peak := [
  ("Mount Nebo",       "USA",     3637, 2013),
  ("Moscow Mountain",  "USA",     1519, 2015),
  ("Himmelbjerget",    "Denmark",  147, 2004),
  ("Mount St. Helens", "USA",     2549, 2010)
]
```

Another example consists of waterfalls and a diary of visits to them:

```
abbrev waterfall : Schema := [
  ⟨"name", .string⟩,
  ⟨"location", .string⟩,
  ⟨"lastVisited", .int⟩
]

def waterfallDiary : Table waterfall := [
  ("Multnomah Falls", "USA", 2018),
  ("Shoshone Falls",  "USA", 2014)
]
```

Recursion and Universes, Revisited

The convenient structuring of rows as tuples comes at a cost: the fact that `Row` treats its two base cases separately means that functions that use `Row` in their types and are defined recursively over the codes (that, is the schema) need to make the same distinctions. One example of a case where this matters is an equality check that uses recursion over the schema to define a function that checks rows for equality. This example does not pass Lean's type checker:

```
def Row.bEq (r1 r2 : Row s) : Bool :=
  match s with
  | [] => true
  | col::cols =>
    match r1, r2 with
    | (v1, r1'), (v2, r2') =>
      v1 == v2 && bEq r1' r2'
```

```
type mismatch
  (v1, r1')
has type
  ?m.3093 × ?m.3096 : Type (max ?u.3105 ?u.3104)
but is expected to have type
  Row (col :: cols) : Type
```

The problem is that the pattern `col :: cols` does not sufficiently refine the type of the rows. This is because Lean cannot yet tell whether the singleton pattern `[col]` or the `col1 :: col2 :: cols` pattern in the definition of `Row` was matched, so the call to `Row` does not compute down to a pair type. The solution is to mirror the structure of `Row` in the definition of `Row.bEq`:

```
def Row.bEq (r1 r2 : Row s) : Bool :=
  match s with
  | [] => true
  | [_] => r1 == r2
  | _::_::_ =>
    match r1, r2 with
    | (v1, r1'), (v2, r2') =>
      v1 == v2 && bEq r1' r2'

instance : BEq (Row s) where
  beq := Row.bEq
```

Unlike in other contexts, functions that occur in types cannot be considered only in terms of their input/output behavior. Programs that use these types will find themselves forced to mirror the algorithm used in the type-level function so that their structure matches the pattern-matching and recursive behavior of the type. A big part of the skill of programming with dependent types is the selection of appropriate type-level functions with the right computational behavior.

Column Pointers

Some queries only make sense if a schema contains a particular column. For example, a query that returns mountains with an elevation greater than 1000 meters only makes sense in the context of a schema with a `"elevation"` column that contains integers. One way to indicate that a column is contained in a schema is to provide a pointer directly to it, and defining the pointer as an indexed family makes it possible to rule out invalid pointers.

There are two ways that a column can be present in a schema: either it is at the beginning of the schema, or it is somewhere later in the schema. Eventually, if a column is later in a schema, then it will be the beginning of some tail of the schema.

The indexed family `HasCol` is a translation of the specification into Lean code:

```
inductive HasCol : Schema → String → DBType → Type where
  | here : HasCol (⟨name, t⟩ :: _) name t
  | there : HasCol s name t → HasCol (_ :: s) name t
```

The family's three arguments are the schema, the column name, and its type. All three are indices, but re-ordering the arguments to place the schema after the column name and type would allow the name and type to be parameters. The constructor `here` can be used when the schema begins with the column `⟨name, t⟩`; it is thus a pointer to the first column in the schema that can only be used when the first column has the desired name and type. The constructor `there` transforms a pointer into a smaller schema into a pointer into a schema with one more column on it.

Because `"elevation"` is the third column in `peak`, it can be found by looking past the first two columns with `there`, after which it is the first column. In other words, to satisfy the type `HasCol peak "elevation" .int`, use the expression `.there (.there .here)`. One way to think about `HasCol` is as a kind of decorated `Nat` — `zero` corresponds to `here`, and `succ` corresponds to `there`. The extra type information makes it impossible to have off-by-one errors.

A pointer to a particular column in a schema can be used to extract that column's value from a row:

```
def Row.get (row : Row s) (col : HasCol s n t) : t.asType :=
  match s, col, row with
  | [_], .here, v => v
  | _::_::_, .here, (v, _) => v
  | _::_::_, .there next, (_, r) => get r next
```

The first step is to pattern match on the schema, because this determines whether the row is a tuple or a single value. No case is needed for the empty schema because there is a `HasCol` available, and both constructors of `HasCol` specify non-empty schemas. If the schema has just a single column, then the pointer must point to it, so only the `here` constructor of `HasCol` need be matched. If the schema has two or more columns, then there must be a case for `here`, in which case the value is the first one in the row, and one for `there`, in which case a recursive call is used. Because the `HasCol` type guarantees that the column exists in the row, `Row.get` does not need to return an `Option`.

`HasCol` plays two roles:

1. It serves as *evidence* that a column with a particular name and type exists in a schema.

2. It serves as *data* that can be used to find the value associated with the column in a row.

The first role, that of evidence, is similar to way that propositions are used. The definition of the indexed family `HasCol` can be read as a specification of what counts as evidence that a given column exists. Unlike propositions, however, it matters which constructor of `HasCol` was used. In the second role, the constructors are used like `Nat`s to find data in a collection. Programming with indexed families often requires the ability to switch fluently between both perspectives.

Subschemas

One important operation in relational algebra is to *project* a table or row into a smaller schema. Every column not present in the smaller schema is forgotten. In order for projection to make sense, the smaller schema must be a subschema of the larger schema, which means that every column in the smaller schema must be present in the larger schema. Just as `HasCol` makes it possible to write a single-column lookup in a row that cannot fail, a representation of the subschema relationship as an indexed family makes it possible to write a projection function that cannot fail.

The ways in which one schema can be a subschema of another can be defined as an indexed family. The basic idea is that a smaller schema is a subschema of a bigger schema if every column in the smaller schema occurs in the bigger schema. If the smaller schema is empty, then it's certainly a subschema of the bigger schema, represented by the constructor `nil`. If the smaller schema has a column, then that column must be in the bigger schema, and all the rest of the columns in the subschema must also be a subschema of the bigger schema. This is represented by the constructor `cons`.

```
inductive Subschema : Schema → Schema → Type where
  | nil : Subschema [] bigger
  | cons :
      HasCol bigger n t →
      Subschema smaller bigger →
      Subschema (⟨n, t⟩ :: smaller) bigger
```

In other words, `Subschema` assigns each column of the smaller schema a `HasCol` that points to its location in the larger schema.

The schema `travelDiary` represents the fields that are common to both `peak` and `waterfall`:

```
abbrev travelDiary : Schema :=
  [⟨"name", .string⟩, ⟨"location", .string⟩, ⟨"lastVisited", .int⟩]
```

It is certainly a subschema of `peak`, as shown by this example:

```
example : Subschema travelDiary peak :=
  .cons .here
    (.cons (.there .here)
      (.cons (.there (.there (.there .here))) .nil))
```

However, code like this is difficult to read and difficult to maintain. One way to improve it is to instruct Lean to write the `Subschema` and `HasCol` constructors automatically. This can be done using the tactic feature that was introduced in the Interlude on propositions and proofs. That interlude uses `by simp` to provide evidence of various propositions.

In this context, two tactics are useful:

- The `constructor` tactic instructs Lean to solve the problem using the constructor of a datatype.
- The `repeat` tactic instructs Lean to repeat a tactic over and over until it either fails or the proof is finished.

In the next example, `by constructor` has the same effect as just writing `.nil` would have:

```
example : Subschema [] peak := by constructor
```

However, attempting that same tactic with a slightly more complicated type fails:

```
example : Subschema [⟨"location", .string⟩] peak := by constructor
```

```
unsolved goals
case a
⊢ HasCol peak "location" DBType.string

case a
⊢ Subschema [] peak
```

Errors that begin with `unsolved goals` describe tactics that failed to completely build the expressions that they were supposed to. In Lean's tactic language, a *goal* is a type that a tactic is to fulfill by constructing an appropriate expression behind the scenes. In this case, `constructor` caused `Subschema.cons` to be applied, and the two goals represent the two arguments expected by `cons`. Adding another instance of `constructor` causes the first goal (`HasCol peak \"location\" DBType.string`) to be addressed with `HasCol.there`, because `peak`'s first column is not `"location"`:

```
example : Subschema [⟨"location", .string⟩] peak := by
  constructor
  constructor
```

```
unsolved goals
case a.a
⊢ HasCol
    [{ name := "location", contains := DBType.string }, { name := "elevation",
contains := DBType.int },
      { name := "lastVisited", contains := DBType.int }]
    "location" DBType.string

case a
⊢ Subschema [] peak
```

However, adding a third `constructor` results in the first goal being solved, because `HasCol.here` is applicable:

```
example : Subschema [⟨"location", .string⟩] peak := by
  constructor
  constructor
  constructor
```

```
unsolved goals
case a
⊢ Subschema [] peak
```

A fourth instance of `constructor` solves the `Subschema peak []` goal:

```
example : Subschema [⟨"location", .string⟩] peak := by
  constructor
  constructor
  constructor
  constructor
```

Indeed, a version written without the use of tactics has four constructors:

```
example : Subschema [⟨"location", .string⟩] peak :=
  .cons (.there .here) .nil
```

Instead of experimenting to find the right number of times to write `constructor`, the `repeat` tactic can be used to ask Lean to just keep trying `constructor` as long as it keeps making progress:

```
example : Subschema [⟨"location", .string⟩] peak := by repeat constructor
```

This more flexible version also works for more interesting `Subschema` problems:

```
example : Subschema travelDiary peak := by repeat constructor

example : Subschema travelDiary waterfall := by repeat constructor
```

The approach of blindly trying constructors until something works is not very useful for types like `Nat` or `List Bool`. Just because an expression has type `Nat` doesn't mean that it's the *correct* `Nat`, after all. But types like `HasCol` and `Subschema` are sufficiently constrained by their indices that only one constructor will ever be applicable, which means that the contents of the program itself are less interesting, and a computer can pick the correct one.

If one schema is a subschema of another, then it is also a subschema of the larger schema extended with an additional column. This fact can be captured as a function definition. `Subschema.addColumn` takes evidence that `smaller` is a subschema of `bigger`, and then returns evidence that `smaller` is a subschema of `c :: bigger`, that is, `bigger` with one additional column:

```
def Subschema.addColumn :
    Subschema smaller bigger →
    Subschema smaller (c :: bigger)
  | .nil  => .nil
  | .cons col sub' => .cons (.there col) sub'.addColumn
```

A subschema describes where to find each column from the smaller schema in the larger schema. `Subschema.addColumn` must translate these descriptions from the original larger schema into the extended larger schema. In the `nil` case, the smaller schema is `[]`, and `nil` is also evidence that `[]` is a subschema of `c :: bigger`. In the `cons` case, which describes how to place one column from `smaller` into `larger`, the placement of the column needs to be adjusted with `there` to account for the new column `c`, and a recursive call adjusts the rest of the columns.

Another way to think about `Subschema` is that it defines a *relation* between two schemas—the existence of an expression with type `Subschema bigger smaller` means that `(bigger, smaller)` is in the relation. This relation is reflexive, meaning that every schema is a subschema of itself:

```
def Subschema.reflexive : (s : Schema) → Subschema s s
  | [] => .nil
  | _ :: cs => .cons .here (reflexive cs).addColumn
```

Projecting Rows

Given evidence that `s'` is a subschema of `s`, a row in `s` can be projected into a row in `s'`. This is done using the evidence that `s'` is a subschema of `s`, which explains where each column of `s'` is found in `s`. The new row in `s'` is built up one column at a time by retrieving the value from the appropriate place in the old row.

The function that performs this projection, `Row.project`, has three cases, one for each case of `Row` itself. It uses `Row.get` together with each `HasCol` in the `Subschema` argument to construct the projected row:

```
def Row.project (row : Row s) : (s' : Schema) → Subschema s' s → Row s'
  | [], .nil => ()
  | [_], .cons c .nil => row.get c
  | _::_::_, .cons c cs => (row.get c, row.project _ cs)
```

Conditions and Selection

Projection removes unwanted columns from a table, but queries must also be able to remove unwanted rows. This operation is called *selection*. Selection relies on having a means of expressing which rows are desired.

The example query language contains expressions, which are analogous to what can be written in a `WHERE` clause in SQL. Expressions are represented by the indexed family `DBExpr`. Because expressions can refer to columns from the database, but different sub-expressions all have the same schema, `DBExpr` takes the database schema as a parameter. Additionally, each expression has a type, and these vary, making it an index:

```
inductive DBExpr (s : Schema) : DBType → Type where
  | col (n : String) (loc : HasCol s n t) : DBExpr s t
  | eq (e1 e2 : DBExpr s t) : DBExpr s .bool
  | lt (e1 e2 : DBExpr s .int) : DBExpr s .bool
  | and (e1 e2 : DBExpr s .bool) : DBExpr s .bool
  | const : t.asType → DBExpr s t
```

The `col` constructor represents a reference to a column in the database. The `eq` constructor compares two expressions for equality, `lt` checks whether one is less than the other, `and` is Boolean conjunction, and `const` is a constant value of some type.

For example, an expression in `peak` that checks whether the `elevation` column is greater than 1000 and the location is `"Denmark"` can be written:

```
def tallInDenmark : DBExpr peak .bool :=
  .and (.lt (.const 1000) (.col "elevation" (by repeat constructor)))
       (.eq (.col "location" (by repeat constructor)) (.const "Denmark"))
```

This is somewhat noisy. In particular, references to columns contain boilerplate calls to `by repeat constructor`. A Lean feature called *macros* can help make expressions easier to read by eliminating this boilerplate:

```
macro "c!" n:term : term => `(DBExpr.col $n (by repeat constructor))
```

This declaration adds the `c!` keyword to Lean, and instructs Lean to replace any instance of `c!` followed by an expression with the corresponding `DBExpr.col` construction. Here, `term` stands for Lean expressions, rather than commands, tactics, or some other part of the language. Lean macros are a bit like C preprocessor macros, except they are better integrated into the language and they automatically avoid some of the pitfalls of CPP. In fact, they are very closely related to macros in Scheme and Racket.

With this macro, the expression can be much easier to read:

```
def tallInDenmark : DBExpr peak .bool :=
  .and (.lt (.const 1000) (c! "elevation"))
       (.eq (c! "location") (.const "Denmark"))
```

Finding the value of an expression with respect to a given row uses `Row.get` to extract column references, and it delegates to Lean's operations on values for every other expression:

```
def DBExpr.evaluate (row : Row s) : DBExpr s t → t.asType
  | .col _ loc => row.get loc
  | .eq e1 e2  => evaluate row e1 == evaluate row e2
  | .lt e1 e2  => evaluate row e1 < evaluate row e2
  | .and e1 e2 => evaluate row e1 && evaluate row e2
  | .const v => v
```

Evaluating the expression for Valby Bakke, the tallest hill in the Copenhagen area, yields `false` because Valby Bakke is much less than 1 km over sea level:

```
#eval tallInDenmark.evaluate ("Valby Bakke", "Denmark", 31, 2023)
```

```
false
```

Evaluating it for a fictional mountain of 1230m elevation yields `true`:

```
#eval tallInDenmark.evaluate ("Fictional mountain", "Denmark", 1230, 2023)
```

```
true
```

Evaluating it for the highest peak in the US state of Idaho yields `false`, as Idaho is not part of Denmark:

```
#eval tallInDenmark.evaluate ("Mount Borah", "USA", 3859, 1996)
```

```
false
```

Queries

The query language is based on relational algebra. In addition to tables, it includes the following operators:

1. The union of two expressions that have the same schema combines the rows that result from two queries
2. The difference of two expressions that have the same schema removes rows found in the second result from the rows in the first result
3. Selection by some criterion filters the result of a query according to an expression
4. Projection into a subschema, removing columns from the result of a query
5. Cartesian product, combining every row from one query with every row from another
6. Renaming a column in the result of a query, which modifies its schema
7. Prefixing all columns in a query with a name

The last operator is not strictly necessary, but it makes the language more convenient to use.

Once again, queries are represented by an indexed family:

```
inductive Query : Schema → Type where
  | table : Table s → Query s
  | union : Query s → Query s → Query s
  | diff : Query s → Query s → Query s
  | select : Query s → DBExpr s .bool → Query s
  | project : Query s → (s' : Schema) → Subschema s' s → Query s'
  | product :
      Query s1 → Query s2 →
      disjoint (s1.map Column.name) (s2.map Column.name) →
      Query (s1 ++ s2)
  | renameColumn :
      Query s → (c : HasCol s n t) → (n' : String) → !((s.map
Column.name).contains n') →
      Query (s.renameColumn c n')
  | prefixWith :
      (n : String) → Query s →
      Query (s.map fun c => {c with name := n ++ "." ++ c.name})
```

The `select` constructor requires that the expression used for selection return a Boolean. The `product` constructor's type contains a call to `disjoint`, which ensures that the two schemas don't share any names:

```
def disjoint [BEq α] (xs ys : List α) : Bool :=
  not (xs.any ys.contains || ys.any xs.contains)
```

The use of an expression of type `Bool` where a type is expected triggers a coercion from `Bool` to `Prop`. Just as decidable propositions can be considered to be Booleans, where evidence for the proposition is coerced to `true` and refutations of the proposition are coerced to `false`, Booleans are coerced into the proposition that states that the expression is equal to `true`. Because all uses of the library are expected to occur in contexts where the schemas are known ahead of time, this proposition can be proved with `by simp`. Similarly, the `renameColumn` constructor checks that the new name does not already exist in the schema. It uses the helper `Schema.renameColumn` to change the name of the column pointed to by `HasCol`:

```
def Schema.renameColumn : (s : Schema) → HasCol s n t → String → Schema
  | c :: cs, .here, n' => {c with name := n'} :: cs
  | c :: cs, .there next, n' => c :: renameColumn cs next n'
```

Executing Queries

Executing queries requires a number of helper functions. The result of a query is a table; this means that each operation in the query language requires a corresponding implementation that works with tables.

Cartesian Product

Taking the Cartesian product of two tables is done by appending each row from the first table to each row from the second. First off, due to the structure of `Row`, adding a single column to a row requires pattern matching on its schema in order to determine whether the result will be a bare value or a tuple. Because this is a common operation, factoring the pattern matching out into a helper is convenient:

```
def addVal (v : c.contains.asType) (row : Row s) : Row (c :: s) :=
  match s, row with
  | [], () => v
  | c' :: cs, v' => (v, v')
```

Appending two rows is recursive on the structure of both the first schema and the first row, because the structure of the row proceeds in lock-step with the structure of the schema. When the first row is empty, appending returns the second row. When the first row is a singleton, the value is added to the second row. When the first row contains multiple columns, the first column's value is added to the result of recursion on the remainder of the row.

```
def Row.append (r1 : Row s1) (r2 : Row s2) : Row (s1 ++ s2) :=
  match s1, r1 with
  | [], () => r2
  | [_], v => addVal v r2
  | _::_::_, (v, r') => (v, r'.append r2)
```

`List.flatMap`, found in the standard library, applies a function that itself returns a list to every entry in an input list, returning the result of appending the resulting lists in order:

```
def List.flatMap (f : α → List β) : (xs : List α) → List β
  | [] => []
  | x :: xs => f x ++ xs.flatMap f
```

The type signature suggests that `List.flatMap` could be used to implement a `Monad List` instance. Indeed, together with `pure x := [x]`, `List.flatMap` does implement a monad. However, it's not a very useful `Monad` instance. The `List` monad is basically a version of `Many` that explores *every* possible path through the search space in advance, before users have the chance to request some number of values. Because of this performance trap, it's usually not a good idea to define a `Monad` instance for `List`. Here, however, the query language has no operator for restricting the number of results to be returned, so combining all possibilities is exactly what is desired:

```
def Table.cartesianProduct (table1 : Table s1) (table2 : Table s2) : Table (s1 +
+ s2) :=
  table1.flatMap fun r1 => table2.map r1.append
```

Just as with `List.product`, a loop with mutation in the identity monad can be used as an alternative implementation technique:

```
def Table.cartesianProduct (table1 : Table s1) (table2 : Table s2) : Table (s1 +
+ s2) := Id.run do
  let mut out : Table (s1 ++ s2) := []
  for r1 in table1 do
    for r2 in table2 do
      out := (r1.append r2) :: out
  pure out.reverse
```

Difference

Removing undesired rows from a table can be done using `List.filter`, which takes a list and a function that returns a `Bool`. A new list is returned that contains only the entries for which the function returns `true`. For instance,

```
["Willamette", "Columbia", "Sandy", "Deschutes"].filter (·.length > 8)
```

evaluates to

```
["Willamette", "Deschutes"]
```

because `"Columbia"` and `"Sandy"` have lengths less than or equal to `8`. Removing the entries of a table can be done using the helper `List.without`:

```
def List.without [BEq α] (source banned : List α) : List α :=
  source.filter fun r => !(banned.contains r)
```

This will be used with the `BEq` instance for `Row` when interpreting queries.

Renaming Columns

Renaming a column in a row is done with a recursive function that traverses the row until the column in question is found, at which point the column with the new name gets the same value as the column with the old name:

```
def Row.rename (c : HasCol s n t) (row : Row s) : Row (s.renameColumn c n') :=
  match s, row, c with
  | [_], v, .here => v
  | _::_::_, (v, r), .here => (v, r)
  | _::_::_, (v, r), .there next => addVal v (r.rename next)
```

While this function changes the *type* of its argument, the actual return value contains precisely the same data as the original argument. From a run-time perspective, `renameRow` is nothing but a slow identity function. One difficulty in programming with indexed families is that when performance matters, this kind of operation can get in the way. It takes a very careful, often brittle, design to eliminate these kinds of "re-indexing" functions.

Prefixing Column Names

Adding a prefix to column names is very similar to renaming a column. Instead of proceeding to a desired column and then returning, `prefixRow` must process all columns:

```
def prefixRow (row : Row s) : Row (s.map fun c => {c with name := n ++ "." ++
c.name}) :=
  match s, row with
  | [], _ => ()
  | [_], v => v
  | _::_::_, (v, r) => (v, prefixRow r)
```

This can be used with `List.map` in order to add a prefix to all rows in a table. Once again, this function only exists to change the type of a value.

Putting the Pieces Together

With all of these helpers defined, executing a query requires only a short recursive function:

```
def Query.exec : Query s → Table s
  | .table t => t
  | .union q1 q2 => exec q1 ++ exec q2
  | .diff q1 q2 => exec q1 |>.without (exec q2)
  | .select q e => exec q |>.filter e.evaluate
  | .project q _ sub => exec q |>.map (·.project _ sub)
  | .product q1 q2 _ => exec q1 |>.cartesianProduct (exec q2)
  | .renameColumn q c _ _ => exec q |>.map (·.rename c)
  | .prefixWith _ q => exec q |>.map prefixRow
```

Some arguments to the constructors are not used during execution. In particular, both the constructor `project` and the function `Row.project` take the smaller schema as explicit arguments, but the type of the *evidence* that this schema is a subschema of the larger schema contains enough information for Lean to fill out the argument automatically. Similarly, the fact that the two tables have disjoint column names that is required by the `product` constructor is not needed by `Table.cartesianProduct`. Generally speaking, dependent types provide many opportunities to have Lean fill out arguments on behalf of the programmer.

Dot notation is used with the results of queries to call functions defined both in the `Table` and `List` namespaces, such `List.map`, `List.filter`, and `Table.cartesianProduct`. This works because `Table` is defined using `abbrev`. Just like type class search, dot notation can see through definitions created with `abbrev`.

The implementation of `select` is also quite concise. After executing the query `q`, `List.filter` is used to remove the rows that do not satisfy the expression. Filter expects a function from `Row s` to `Bool`, but `DBExpr.evaluate` has type `Row s → DBExpr s t → t.asType`. Because the type of the `select` constructor requires that the expression have type `DBExpr s .bool`, `t.asType` is actually `Bool` in this context.

A query that finds the heights of all mountain peaks with an elevation greater than 500 meters can be written:

```
open Query in
def example1 :=
  table mountainDiary |>.select
  (.lt (.const 500) (c! "elevation")) |>.project
  [⟨"elevation", .int⟩] (by repeat constructor)
```

Executing it returns the expected list of integers:

```
#eval example1.exec
```

```
[3637, 1519, 2549]
```

To plan a sightseeing tour, it may be relevant to match all pairs mountains and waterfalls in the same location. This can be done by taking the Cartesian product of both tables, selecting only the rows in which they are equal, and then projecting out the names:

```
open Query in
def example2 :=
  let mountain := table mountainDiary |>.prefixWith "mountain"
  let waterfall := table waterfallDiary |>.prefixWith "waterfall"
  mountain.product waterfall (by decide)
    |>.select (.eq (c! "mountain.location") (c! "waterfall.location"))
    |>.project [⟨"mountain.name", .string⟩, ⟨"waterfall.name", .string⟩] (by
repeat constructor)
```

Because the example data includes only waterfalls in the USA, executing the query returns pairs of mountains and waterfalls in the US:

```
#eval example2.exec
```

```
[("Mount Nebo", "Multnomah Falls"), ("Mount Nebo", "Shoshone Falls"), ("Moscow
Mountain", "Multnomah Falls"),
  ("Moscow Mountain", "Shoshone Falls"), ("Mount St. Helens", "Multnomah
Falls"),
  ("Mount St. Helens", "Shoshone Falls")]
```

Errors You May Meet

Many potential errors are ruled out by the definition of `Query`. For instance, forgetting the added qualifier in `"mountain.location"` yields a compile-time error that highlights the column reference `c! "location"`:

```
open Query in
def example2 :=
  let mountains := table mountainDiary |>.prefixWith "mountain"
  let waterfalls := table waterfallDiary |>.prefixWith "waterfall"
  mountains.product waterfalls (by simp)
    |>.select (.eq (c! "location") (c! "waterfall.location"))
    |>.project [⟨"mountain.name", .string⟩, ⟨"waterfall.name", .string⟩] (by
repeat constructor)
```

This is excellent feedback! On the other hand, the text of the error message is quite difficult to act on:

```
unsolved goals
case a.a.a.a.a.a.a.a
mountains : Query (List.map (fun c => { name := "mountain" ++ "." ++ c.name,
contains := c.contains }) peak) :=
  prefixWith "mountain" (table mountainDiary)
waterfalls : Query (List.map (fun c => { name := "waterfall" ++ "." ++ c.name,
contains := c.contains }) waterfall) :=
  prefixWith "waterfall" (table waterfallDiary)
⊢ HasCol (List.map (fun c => { name := "waterfall" ++ "." ++ c.name, contains :=
c.contains }) []) "location" ?m.52988
```

Similarly, forgetting to add prefixes to the names of the two tables results in an error on `by simp`, which should provide evidence that the schemas are in fact disjoint;

```
open Query in
def example2 :=
  let mountains := table mountainDiary
  let waterfalls := table waterfallDiary
  mountains.product waterfalls (by decide)
    |>.select (.eq (c! "mountain.location") (c! "waterfall.location"))
    |>.project [⟨"mountain.name", .string⟩, ⟨"waterfall.name", .string⟩] (by
repeat constructor)
```

However, the error message is similarly unhelpful:

```
tactic 'decide' proved that the proposition
  disjoint (List.map Column.name peak) (List.map Column.name waterfall) = true
is false
```

Lean's macro system contains everything needed not only to provide a convenient syntax for queries, but also to arrange for the error messages to be helpful. Unfortunately, it is beyond the scope of this book to provide a description of implementing languages with Lean macros. An indexed family such as `Query` is probably best as the core of a typed database interaction library, rather than its user interface.

Exercises

Dates

Define a structure to represent dates. Add it to the `DBType` universe and update the rest of the code accordingly. Provide the extra `DBExpr` constructors that seem to be necessary.

Nullable Types

Add support for nullable columns to the query language by representing database types with the following structure:

```
structure NDBType where
  underlying : DBType
  nullable : Bool

abbrev NDBType.asType (t : NDBType) : Type :=
  if t.nullable then
    Option t.underlying.asType
  else
    t.underlying.asType
```

Use this type in place of `DBType` in `Column` and `DBExpr`, and look up SQL's rules for `NULL` and comparison operators to determine the types of `DBExpr`'s constructors.

Experimenting with Tactics

What is the result of asking Lean to find values of the following types using `by repeat constructor`? Explain why each gives the result that it does.

- `Nat`
- `List Nat`
- `Vect Nat 4`
- `Row []`
- `Row [⟨"price", .int⟩]`
- `Row peak`
- `HasCol [⟨"price", .int⟩, ⟨"price", .int⟩] "price" .int`

Indices, Parameters, and Universe Levels

The distinction between indices and parameters of an inductive type is more than just a way to describe arguments to the type that either vary or do not between the constructors. Whether an argument to an inductive type is a parameter or an index also matters when it comes time to determine the relationships between their universe levels. In particular, an inductive type may have the same universe level as a parameter, but it must be in a larger universe than its indices. This restriction is necessary to ensure that Lean can be used as a theorem prover as well as a programming language—without it, Lean's logic would be inconsistent. Experimenting with error messages is a good way to illustrate these rules, as well as the precise rules that determine whether an argument to a type is a parameter or an index.

Generally speaking, the definition of an inductive type takes its parameters before a colon and its indices after the colon. Parameters are given names like function arguments, whereas indices only have their types described. This can be seen in the definition of `Vect`:

```
inductive Vect (α : Type u) : Nat → Type u where
   | nil : Vect α 0
   | cons : α → Vect α n → Vect α (n + 1)
```

In this definition, `α` is a parameter and the `Nat` is an index. Parameters may be referred to throughout the definition (for example, `Vect.cons` uses `α` for the type of its first argument), but they must always be used consistently. Because indices are expected to change, they are assigned individual values at each constructor, rather than being provided as arguments at the top of the datatype definition.

A very simple datatype with a parameter is `WithParameter`:

```
inductive WithParameter (α : Type u) : Type u where
  | test : α → WithParameter α
```

The universe level `u` can be used for both the parameter and for the inductive type itself, illustrating that parameters do not increase the universe level of a datatype. Similarly, when there are multiple parameters, the inductive type receives whichever universe level is greater:

```
inductive WithTwoParameters (α : Type u) (β : Type v) : Type (max u v) where
  | test : α → β → WithTwoParameters α β
```

Because parameters do not increase the universe level of a datatype, they can be more convenient to work with. Lean attempts to identify arguments that are described like indices (after the colon), but used like parameters, and turn them into parameters: Both of the following inductive datatypes have their parameter written after the colon:

```
inductive WithParameterAfterColon : Type u → Type u where
  | test : α → WithParameterAfterColon α

inductive WithParameterAfterColon2 : Type u → Type u where
  | test1 : α → WithParameterAfterColon2 α
  | test2 : WithParameterAfterColon2 α
```

When a parameter is not named in the initial datatype declaration, different names may be used for it in each constructor, so long as they are used consistently. The following declaration is accepted:

```
inductive WithParameterAfterColonDifferentNames : Type u → Type u where
  | test1 : α → WithParameterAfterColonDifferentNames α
  | test2 : β → WithParameterAfterColonDifferentNames β
```

However, this flexibility does not extend to datatypes that explicitly declare the names of their parameters:

```
inductive WithParameterBeforeColonDifferentNames (α : Type u) : Type u where
  | test1 : α → WithParameterBeforeColonDifferentNames α
  | test2 : β → WithParameterBeforeColonDifferentNames β
```

```
inductive datatype parameter mismatch
  β
expected
  α
```

Similarly, attempting to name an index results in an error:

```
inductive WithNamedIndex (α : Type u) : Type (u + 1) where
  | test1 : WithNamedIndex α
  | test2 : WithNamedIndex α → WithNamedIndex α → WithNamedIndex (α × α)
```

```
inductive datatype parameter mismatch
  α × α
expected
  α
```

Using an appropriate universe level and placing the index after the colon results in a declaration that is acceptable:

```
inductive WithIndex : Type u → Type (u + 1) where
  | test1 : WithIndex α
  | test2 : WithIndex α → WithIndex α → WithIndex (α × α)
```

Even though Lean can sometimes determine that an argument after the colon in an inductive type declaration is a parameter when it is used consistently in all constructors, all parameters are still required to come before all indices. Attempting to place a parameter after an index results in the argument being considered an index itself, which would require the universe level of the datatype to increase:

```
inductive ParamAfterIndex : Nat → Type u → Type u where
  | test1 : ParamAfterIndex 0 γ
  | test2 : ParamAfterIndex n γ → ParamAfterIndex k γ → ParamAfterIndex (n + k)
γ
```

```
invalid universe level in constructor 'ParamAfterIndex.test1', parameter 'γ' has
type
  Type u
at universe level
  u+2
which is not less than or equal to the inductive type's resulting universe level
  u+1
```

Parameters need not be types. This example shows that ordinary datatypes such as `Nat` may be used as parameters:

```
inductive NatParam (n : Nat) : Nat → Type u where
  | five : NatParam 4 5
```

```
inductive datatype parameter mismatch
  4
expected
  n
```

Using the `n` as suggested causes the declaration to be accepted:

```
inductive NatParam (n : Nat) : Nat → Type u where
  | five : NatParam n 5
```

What can be concluded from these experiments? The rules of parameters and indices are as follows:

1. Parameters must be used identically in each constructor's type.
2. All parameters must come before all indices.
3. The universe level of the datatype being defined must be at least as large as the largest

parameter, and strictly larger than the largest index.

4. Named arguments written before the colon are always parameters, while arguments after the colon are typically indices. Lean may determine that the usage of arguments after the colon makes them into parameters if they are used consistently in all constructors and don't come after any indices.

When in doubt, the Lean command `#print` can be used to check how many of a datatype's arguments are parameters. For example, for `Vect`, it points out that the number of parameters is 1:

```
#print Vect
```

```
inductive Vect.{u} : Type u → Nat → Type u
number of parameters: 1
constructors:
Vect.nil : {α : Type u} → Vect α 0
Vect.cons : {α : Type u} → {n : Nat} → α → Vect α n → Vect α (n + 1)
```

It is worth thinking about which arguments should be parameters and which should be indices when choosing the order of arguments to a datatype. Having as many arguments as possible be parameters helps keep universe levels under control, which can make a complicated program easier to type check. One way to make this possible is to ensure that all parameters come before all indices in the argument list.

Additionally, even though Lean is capable of determining that arguments after the colon are nonetheless parameters by their usage, it's a good idea to write parameters with explicit names. This makes the intention clear to readers, and it causes Lean to report an error if the argument is mistakenly used inconsistently across the constructors.

Pitfalls of Programming with Dependent Types

The flexibility of dependent types allows more useful programs to be accepted by a type checker, because the language of types is expressive enough to describe variations that less-expressive type systems cannot. At the same time, the ability of dependent types to express very fine-grained specifications allows more buggy programs to be rejected by a type checker. This power comes at a cost.

The close coupling between the internals of type-returning functions such as `Row` and the types that they produce is an instance of a bigger difficulty: the distinction between the interface and the implementation of functions begins to break down when functions are used in types. Normally, all refactorings are valid as long as they don't change the type signature or input-output behavior of a function. Functions can be rewritten to use more efficient algorithms and data structures, bugs can be fixed, and code clarity can be improved without breaking client code. When the function is used in a type, however, the internals of the function's implementation become part of the type, and thus part of the *interface* to another program.

As an example, take the following two implementations of addition on `Nat`. `Nat.plusL` is recursive on its first argument:

```
def Nat.plusL : Nat → Nat → Nat
  | 0, k => k
  | n + 1, k => plusL n k + 1
```

`Nat.plusR`, on the other hand, is recursive on its second argument:

```
def Nat.plusR : Nat → Nat → Nat
  | n, 0 => n
  | n, k + 1 => plusR n k + 1
```

Both implementations of addition are faithful to the underlying mathematical concept, and they thus return the same result when given the same arguments.

However, these two implementations present quite different interfaces when they are used in types. As an example, take a function that appends two `Vect`s. This function should return a `Vect` whose length is the sum of the length of the arguments. Because `Vect` is essentially a `List` with a more informative type, it makes sense to write the function just as one would for `List.append`, with pattern matching and recursion on the first argument. Starting with a type signature and initial pattern match pointing at placeholders yields two messages:

```
def appendL : Vect α n → Vect α k → Vect α (n.plusL k)
  | .nil, ys => _
  | .cons x xs, ys => _
```

The first message, in the `nil` case, states that the placeholder should be replaced by a `Vect` with length `plusL 0 k`:

```
don't know how to synthesize placeholder
context:
α : Type u_1
n k : Nat
ys : Vect α k
⊢ Vect α (Nat.plusL 0 k)
```

The second message, in the `cons` case, states that the placeholder should be replaced by a `Vect` with length `plusL (n✝ + 1) k`:

```
don't know how to synthesize placeholder
context:
α : Type u_1
n k n✝ : Nat
x : α
xs : Vect α n✝
ys : Vect α k
⊢ Vect α ((n✝ + 1).plusL k)
```

The symbol after `n`, called a *dagger*, is used to indicate names that Lean has internally invented. Behind the scenes, pattern matching on the first `Vect` implicitly caused the value of the first `Nat` to be refined as well, because the index on the constructor `cons` is `n + 1`, with the tail of the `Vect` having length `n`. Here, `n✝` represents the `Nat` that is one less than the argument `n`.

Definitional Equality

In the definition of `plusL`, there is a pattern case `0, k => k`. This applies in the length used in the first placeholder, so another way to write the underscore's type `Vect α (Nat.plusL 0 k)` is `Vect α k`. Similarly, `plusL` contains a pattern case `n + 1, k => plusN n k + 1`. This means that the type of the second underscore can be equivalently written `Vect α (plusL n✝ k + 1)`.

To expose what is going on behind the scenes, the first step is to write the `Nat` arguments explicitly, which also results in daggerless error messages because the names are now written explicitly in the program:

```
def appendL : (n k : Nat) → Vect α n → Vect α k → Vect α (n.plusL k)
  | 0, k, .nil, ys => _
  | n + 1, k, .cons x xs, ys => _
```

```
don't know how to synthesize placeholder
context:
α : Type u_1
k : Nat
ys : Vect α k
⊢ Vect α (Nat.plusL 0 k)
```

```
don't know how to synthesize placeholder
context:
α : Type u_1
n k : Nat
x : α
xs : Vect α n
ys : Vect α k
⊢ Vect α ((n + 1).plusL k)
```

Annotating the underscores with the simplified versions of the types does not introduce a type error, which means that the types as written in the program are equivalent to the ones that Lean found on its own:

```
def appendL : (n k : Nat) → Vect α n → Vect α k → Vect α (n.plusL k)
  | 0, k, .nil, ys => (_ : Vect α k)
  | n + 1, k, .cons x xs, ys => (_ : Vect α (n.plusL k + 1))
```

```
don't know how to synthesize placeholder
context:
α : Type u_1
k : Nat
ys : Vect α k
⊢ Vect α k
```

```
don't know how to synthesize placeholder
context:
α : Type u_1
n k : Nat
x : α
xs : Vect α n
ys : Vect α k
⊢ Vect α (n.plusL k + 1)
```

The first case demands a `Vect α k`, and `ys` has that type. This is parallel to the way that appending the empty list to any other list returns that other list. Refining the definition with `ys` instead of the first underscore yields a program with only one remaining underscore to be filled out:

```
def appendL : (n k : Nat) → Vect α n → Vect α k → Vect α (n.plusL k)
  | 0, k, .nil, ys => ys
  | n + 1, k, .cons x xs, ys => (_ : Vect α (n.plusL k + 1))
```

```
don't know how to synthesize placeholder
context:
α : Type u_1
n k : Nat
x : α
xs : Vect α n
ys : Vect α k
⊢ Vect α (n.plusL k + 1)
```

Something very important has happened here. In a context where Lean expected a `Vect α (Nat.plusL 0 k)`, it received a `Vect α k`. However, `Nat.plusL` is not an `abbrev`, so it may seem like it shouldn't be running during type checking. Something else is happening.

The key to understanding what's going on is that Lean doesn't just expand `abbrev`s while type checking. It can also perform computation while checking whether two types are equivalent to one another, such that any expression of one type can be used in a context that expects the other type. This property is called *definitional equality*, and it is subtle.

Certainly, two types that are written identically are considered to be definitionally equal—`Nat` and `Nat` or `List String` and `List String` should be considered equal. Any two concrete types built from different datatypes are not equal, so `List Nat` is not equal to `Int`. Additionally, types that differ only by renaming internal names are equal, so `(n : Nat) → Vect String n` is the same as `(k : Nat) → Vect String k`. Because types can contain ordinary data, definitional equality must also describe when data are equal. Uses of the same constructors are equal, so `0` equals `0` and `[5, 3, 1]` equals `[5, 3, 1]`.

Types contain more than just function arrows, datatypes, and constructors, however. They also contain *variables* and *functions*. Definitional equality of variables is relatively simple: each variable is equal only to itself, so `(n k : Nat) → Vect Int n` is not definitionally equal to `(n k : Nat) → Vect Int k`. Functions, on the other hand, are more complicated. While mathematics considers two functions to be equal if they have identical input-output behavior, there is no efficient algorithm to check that, and the whole point of definitional equality is for Lean to check whether two types are interchangeable. Instead, Lean considers functions to be definitionally equal either when they are both `fun`-expressions with definitionally equal bodies. In other words, two functions must use *the same algorithm* that calls *the same helpers* to be considered definitionally equal. This is not typically very helpful, so definitional equality of functions is mostly used when the exact same defined function occurs in two types.

When functions are *called* in a type, checking definitional equality may involve reducing the function call. The type `Vect String (1 + 4)` is definitionally equal to the type `Vect String (3 + 2)` because `1 + 4` is definitionally equal to `3 + 2`. To check their equality, both are reduced to `5`, and then the constructor rule can be used five times. Definitional equality of functions applied to data can be checked first by seeing if they're already the same—there's no need to reduce `["a", "b"] ++ ["c"]` to check that it's equal to `["a", "b"] ++ ["c"]`, after all. If not, the function is called and replaced with its value, and the value can then be checked.

Not all function arguments are concrete data. For example, types may contain `Nat`s that are not built from the `zero` and `succ` constructors. In the type `(n : Nat) → Vect String n`, the variable `n` is a `Nat`, but it is impossible to know *which* `Nat` it is before the function is called. Indeed, the function may be called first with `0`, and then later with `17`, and then again with `33`. As seen in the definition of `appendL`, variables with type `Nat` may also be passed to functions such as `plusL`. Indeed, the type `(n : Nat) → Vect String n` is definitionally equal to the type `(n : Nat) → Vect String (Nat.plusL 0 n)`.

The reason that `n` and `Nat.plusL 0 n` are definitionally equal is that `plusL`'s pattern match examines its *first* argument. This is problematic: `(n : Nat) → Vect String n` is *not* definitionally equal to `(n : Nat) → Vect String (Nat.plusL n 0)`, even though zero should be both a left and a right identity of addition. This happens because pattern matching gets stuck when it encounters variables. Until the actual value of `n` becomes known, there is no way to know which case of `Nat.plusL n 0` should be selected.

The same issue appears with the `Row` function in the query example. The type `Row (c :: cs)` does not reduce to any datatype because the definition of `Row` has separate cases for singleton lists and lists with at least two entries. In other words, it gets stuck when trying to match the variable `cs` against concrete `List` constructors. This is why almost every function that takes apart or constructs a `Row` needs to match the same three cases as `Row` itself: getting it unstuck reveals concrete types that can be used for either pattern matching or constructors.

The missing case in `appendL` requires a `Vect α (Nat.plusL n k + 1)`. The `+ 1` in the index suggests that the next step is to use `Vect.cons`:

```
def appendL : (n k : Nat) → Vect α n → Vect α k → Vect α (n.plusL k)
  | 0, k, .nil, ys => ys
  | n + 1, k, .cons x xs, ys => .cons x (_ : Vect α (n.plusL k))
```

```
don't know how to synthesize placeholder
context:
α : Type u_1
n k : Nat
x : α
xs : Vect α n
ys : Vect α k
⊢ Vect α (n.plusL k)
```

A recursive call to `appendL` can construct a `Vect` with the desired length:

```
def appendL : (n k : Nat) → Vect α n → Vect α k → Vect α (n.plusL k)
  | 0, k, .nil, ys => ys
  | n + 1, k, .cons x xs, ys => .cons x (appendL n k xs ys)
```

Now that the program is finished, removing the explicit matching on `n` and `k` makes it easier to read and easier to call the function:

```
def appendL : Vect α n → Vect α k → Vect α (n.plusL k)
  | .nil, ys => ys
  | .cons x xs, ys => .cons x (appendL xs ys)
```

Comparing types using definitional equality means that everything involved in definitional equality, including the internals of function definitions, becomes part of the *interface* of programs that use dependent types and indexed families. Exposing the internals of a function in a type means that refactoring the exposed program may cause programs that use it to no longer type check. In particular, the fact that `plusL` is used in the type of `appendL` means that the definition of `plusL` cannot be replaced by the otherwise-equivalent `plusR`.

Getting Stuck on Addition

What happens if append is defined with `plusR` instead? Beginning in the same way, with explicit lengths and placeholder underscores in each case, reveals the following useful error messages:

```
def appendR : (n k : Nat) → Vect α n → Vect α k → Vect α (n.plusR k)
  | 0, k, .nil, ys => _
  | n + 1, k, .cons x xs, ys => _
```

```
don't know how to synthesize placeholder
context:
α : Type u_1
k : Nat
ys : Vect α k
⊢ Vect α (Nat.plusR 0 k)
```

```
don't know how to synthesize placeholder
context:
α : Type u_1
n k : Nat
x : α
xs : Vect α n
ys : Vect α k
⊢ Vect α ((n + 1).plusR k)
```

However, attempting to place a `Vect α k` type annotation around the first placeholder results in an type mismatch error:

```
def appendR : (n k : Nat) → Vect α n → Vect α k → Vect α (n.plusR k)
  | 0, k, .nil, ys => (_ : Vect α k)
  | n + 1, k, .cons x xs, ys => _
```

```
type mismatch
  ?m.2199
has type
  Vect α k : Type ?u.2135
but is expected to have type
  Vect α (Nat.plusR 0 k) : Type ?u.2135
```

This error is pointing out that `plusR 0 k` and `k` are *not* definitionally equal.

This is because `plusR` has the following definition:

```
def Nat.plusR : Nat → Nat → Nat
  | n, 0 => n
  | n, k + 1 => plusR n k + 1
```

Its pattern matching occurs on the *second* argument, not the first argument, which means that the presence of the variable `k` in that position prevents it from reducing. `Nat.add` in Lean's standard library is equivalent to `plusR`, not `plusL`, so attempting to use it in this definition results in precisely the same difficulties:

```
def appendR : (n k : Nat) → Vect α n → Vect α k → Vect α (n + k)
  | 0, k, .nil, ys => (_ : Vect α k)
  | n + 1, k, .cons x xs, ys => _
```

```
type mismatch
  ?m.2234
has type
  Vect α k : Type ?u.2135
but is expected to have type
  Vect α (0 + k) : Type ?u.2135
```

Addition is getting *stuck* on the variables. Getting it unstuck requires propositional equality.

Propositional Equality

Propositional equality is the mathematical statement that two expressions are equal. While definitional equality is a kind of ambient fact that Lean automatically checks when required, statements of propositional equality require explicit proofs. Once an equality proposition has been proved, it can be used in a program to modify a type, replacing one side of the equality with the other, which can unstick the type checker.

The reason why definitional equality is so limited is to enable it to be checked by an algorithm. Propositional equality is much richer, but the computer cannot in general check whether two expressions are propositionally equal, though it can verify that a purported proof is in fact a proof. The split between definitional and propositional equality represents a division of labor between humans and machines: the most boring equalities are checked automatically as part of definitional equality, freeing the human mind to work on the interesting problems available in propositional equality. Similarly, definitional equality is invoked automatically by the type checker, while propositional equality must be specifically appealed to.

In Propositions, Proofs, and Indexing, some equality statements are proved using `decide`. All of these equality statements are ones in which the propositional equality is in fact already a definitional equality. Typically, statements of propositional equality are proved by first getting them into a form where they are either definitional or close enough to existing proved equalities, and then using tools like `decide` or `simp` to take care of the simplified cases. The `simp` tactic is quite powerful: behind the scenes, it uses a number of fast, automated tools to construct a proof. A simpler tactic called `rfl` specifically uses definitional equality to prove propositional equality. The name `rfl` is short for *reflexivity*, which is the property of equality that states that everything equals itself.

Unsticking `appendR` requires a proof that `k = Nat.plusR 0 k`, which is not a definitional equality because `plusR` is stuck on the variable in its second argument. To get it to compute, the `k` must become a concrete constructor. This is a job for pattern matching.

In particular, because `k` could be *any* `Nat`, this task requires a function that can return evidence that `k = Nat.plusR 0 k` for *any* `k` whatsoever. This should be a function that returns a proof of equality, with type `(k : Nat) → k = Nat.plusR 0 k`. Getting it started with initial patterns and placeholders yields the following messages:

```
def plusR_zero_left : (k : Nat) → k = Nat.plusR 0 k
  | 0 => _
  | k + 1 => _
```

```
don't know how to synthesize placeholder
context:
⊢ 0 = Nat.plusR 0 0
```

```
don't know how to synthesize placeholder
context:
k : Nat
⊢ k + 1 = Nat.plusR 0 (k + 1)
```

Having refined `k` to `0` via pattern matching, the first placeholder stands for evidence of a statement that does hold definitionally. The `rfl` tactic takes care of it, leaving only the second placeholder:

```
def plusR_zero_left : (k : Nat) → k = Nat.plusR 0 k
  | 0 => by rfl
  | k + 1 => _
```

The second placeholder is a bit trickier. The expression `Nat.plusR 0 k + 1` is definitionally equal to `Nat.plusR 0 (k + 1)`. This means that the goal could also be written `k + 1 = Nat.plusR 0 k + 1`:

```
def plusR_zero_left : (k : Nat) → k = Nat.plusR 0 k
  | 0 => by rfl
  | k + 1 => (_ : k + 1 = Nat.plusR 0 k + 1)
```

```
don't know how to synthesize placeholder
context:
k : Nat
⊢ k + 1 = Nat.plusR 0 k + 1
```

Underneath the `+ 1` on each side of the equality statement is another instance of what the function itself returns. In other words, a recursive call on `k` would return evidence that `k = Nat.plusR 0 k`. Equality wouldn't be equality if it didn't apply to function arguments. In other words, if `x = y`, then `f x = f y`. The standard library contains a function `congrArg` that takes a function and an equality proof and returns a new proof where the function has been applied to both sides of the equality. In this case, the function is `(· + 1)`:

```
def plusR_zero_left : (k : Nat) → k = Nat.plusR 0 k
  | 0 => by rfl
  | k + 1 =>
    congrArg (· + 1) (plusR_zero_left k)
```

Because this is really a proof of a proposition, it should be declared as a `theorem`:

```
theorem plusR_zero_left : (k : Nat) → k = Nat.plusR 0 k
  | 0 => by rfl
  | k + 1 =>
    congrArg (· + 1) (plusR_zero_left k)
```

Propositional equalities can be deployed in a program using the rightward triangle operator `▸`. Given an equality proof as its first argument and some other expression as its second, this operator replaces instances of one side of the equality with the other side of the equality in the second argument's type. In other words, the following definition contains no type errors:

```
def appendR : (n k : Nat) → Vect α n → Vect α k → Vect α (n.plusR k)
  | 0, k, .nil, ys => plusR_zero_left k ▸ (_ : Vect α k)
  | n + 1, k, .cons x xs, ys => _
```

The first placeholder has the expected type:

```
don't know how to synthesize placeholder
context:
α : Type u_1
k : Nat
ys : Vect α k
⊢ Vect α k
```

It can now be filled in with `ys`:

```
def appendR : (n k : Nat) → Vect α n → Vect α k → Vect α (n.plusR k)
  | 0, k, .nil, ys => plusR_zero_left k ▸ ys
  | n + 1, k, .cons x xs, ys => _
```

Filling in the remaining placeholder requires unsticking another instance of addition:

```
don't know how to synthesize placeholder
context:
α : Type u_1
n k : Nat
x : α
xs : Vect α n
ys : Vect α k
⊢ Vect α ((n + 1).plusR k)
```

Here, the statement to be proved is that `Nat.plusR (n + 1) k = Nat.plusR n k + 1`, which can be used with `▸` to draw the `+ 1` out to the top of the expression so that it matches the index of `cons`.

The proof is a recursive function that pattern matches on the second argument to `plusR`, namely `k`. This is because `plusR` itself pattern matches on its second argument, so the proof can "unstick" it through pattern matching, exposing the computational behavior. The skeleton of the proof is very similar to that of `plusR_zero_left`:

```
theorem plusR_succ_left (n : Nat) : (k : Nat) → Nat.plusR (n + 1) k = Nat.plusR
n k + 1
  | 0 => by rfl
  | k + 1 => _
```

The remaining case's type is definitionally equal to `Nat.plusR (n + 1) k + 1 = Nat.plusR n (k + 1) + 1`, so it can be solved with `congrArg`, just as in `plusR_zero_left`:

```
don't know how to synthesize placeholder
context:
n k : Nat
⊢ (n + 1).plusR (k + 1) = n.plusR (k + 1) + 1
```

This results in a finished proof:

```
theorem plusR_succ_left (n : Nat) : (k : Nat) → Nat.plusR (n + 1) k = Nat.plusR
n k + 1
  | 0 => by rfl
  | k + 1 => congrArg (· + 1) (plusR_succ_left n k)
```

The finished proof can be used to unstick the second case in `appendR`:

```
def appendR : (n k : Nat) → Vect α n → Vect α k → Vect α (n.plusR k)
  | 0, k, .nil, ys => plusR_zero_left k ▸ ys
  | n + 1, k, .cons x xs, ys => plusR_succ_left n k ▸ .cons x (appendR n k xs
ys)
```

When making the length arguments to `appendR` implicit again, they are no longer explicitly named to be appealed to in the proofs. However, Lean's type checker has enough information to fill them in automatically behind the scenes, because no other values would allow the types to match:

```
def appendR : Vect α n → Vect α k → Vect α (n.plusR k)
  | .nil, ys => plusR_zero_left _ ▸ ys
  | .cons x xs, ys => plusR_succ_left _ _ ▸ .cons x (appendR xs ys)
```

Pros and Cons

Indexed families have an important property: pattern matching on them affects definitional equality. For example, in the `nil` case in a `match` expression on a `Vect`, the length simply *becomes* `0`. Definitional equality can be very convenient, because it is always active and does not need to be invoked explicitly.

However, the use of definitional equality with dependent types and pattern matching has serious software engineering drawbacks. First off, functions must be written especially to be used in types, and functions that are convenient to use in types may not use the most efficient algorithms. Once a function has been exposed through using it in a type, its implementation has become part of the interface, leading to difficulties in future refactoring. Secondly, definitional equality can be slow. When asked to check whether two expressions are definitionally equal, Lean may need to run large amounts of code if the functions in question are complicated and have many layers of abstraction. Third, error messages that result from failures of definitional equality are not always very easy to understand, because they may be phrased in terms of the internals of functions. It is not always easy to understand the provenance of the expressions in the error messages. Finally, encoding non-trivial invariants in a collection of indexed families and dependently-typed functions can often be brittle. It is often necessary to change early definitions in a system when the exposed reduction behavior of functions proves to not provide convenient definitional equalities. The alternative is to litter the program with appeals to equality proofs, but these can become quite unwieldy.

In idiomatic Lean code, indexed datatypes are not used very often. Instead, subtypes and explicit propositions are typically used to enforce important invariants. This approach involves many explicit proofs, and very few appeals to definitional equality. As befits an interactive theorem prover, Lean has been designed to make explicit proofs convenient. Generally speaking, this approach should be preferred in most cases.

However, understanding indexed families of datatypes is important. Recursive functions such as `plusR_zero_left` and `plusR_succ_left` are in fact *proofs by mathematical* *induction*. The base case of the recursion corresponds to the base case in induction, and the recursive call represents an appeal to the induction hypothesis. More generally, new propositions in Lean are often defined as inductive types of evidence, and these inductive types usually have indices. The process of proving theorems is in fact constructing expressions with these types behind the scenes, in a process not unlike the proofs in this section. Also, indexed datatypes are sometimes exactly the right tool for the job. Fluency in their use is an important part of knowing when to use them.

Exercises

- Using a recursive function in the style of `plusR_succ_left`, prove that for all `Nat`s `n` and `k`, `n.plusR k = n + k`.
- Write a function on `Vect` for which `plusR` is more natural than `plusL`, where `plusL` would require proofs to be used in the definition.

Summary

Dependent Types

Dependent types, where types contain non-type code such as function calls and ordinary data constructors, lead to a massive increase in the expressive power of a type system. The ability to *compute* a type from the *value* of an argument means that the return type of a function can vary based on which argument is provided. This can be used, for example, to have the result type of a database query depend on the database's schema and the specific query issued, without needing any potentially-failing cast operations on the result of the query. When the query changes, so does the type that results from running it, enabling immediate compile-time feedback.

When a function's return type depends on a value, analyzing the value with pattern matching can result in the type being *refined*, as a variable that stands for a value is replaced by the constructors in the pattern. The type signature of a function documents the way that the return type depends on the argument value, and pattern matching then explains how the return type can be fulfilled for each potential argument.

Ordinary code that occurs in types is run during type checking, though `partial` functions that might loop infinitely are not called. Mostly, this computation follows the rules of ordinary evaluation that were introduced in the very beginning of this book, with expressions being progressively replaced by their values until a final value is found. Computation during type checking has an important difference from run-time computation: some values in types may be *variables* whose values are not yet known. In these cases, pattern-matching gets "stuck" and does not proceed until or unless a particular constructor is selected, e.g. by pattern matching. Type-level computation can be seen as a kind of partial evaluation, where only the parts of the program that are sufficiently known need to be evaluated and other parts are left alone.

The Universe Pattern

A common pattern when working with dependent types is to section off some subset of the type system. For example, a database query library might be able to return varying-length strings, fixed-length strings, or numbers in certain ranges, but it will never return a function, a user-defined datatype, or an `IO` action. A domain-specific subset of the type system can be defined by first defining a datatype with constructors that match the structure of the desired types, and then defining a function that interprets values from this datatype into honest-to-goodness types. The constructors are referred to as *codes* for the types in question, and the entire pattern is sometimes referred to as a *universe à la Tarski*, or just as a *universe* when context makes it clear that universes such as `Type 3` or `Prop` are not what's meant.

Custom universes are an alternative to defining a type class with instances for each type of interest. Type classes are extensible, but extensibility is not always desired. Defining a custom universe has a number of advantages over using the types directly:

- Generic operations that work for *any* type in the universe, such as equality testing and serialization, can be implemented by recursion on codes.
- The types accepted by external systems can be represented precisely, and the definition of the code datatype serves to document what can be expected.
- Lean's pattern matching completeness checker ensures that no codes are forgotten, while solutions based on type classes defer missing instance errors to client code.

Indexed Families

Datatypes can take two separate kinds of arguments: *parameters* are identical in each constructor of the datatype, while *indices* may vary between constructors. For a given choice of index, only some constructors of the datatype are available. As an example, `Vect.nil` is available only when the length index is `0`, and `Vect.cons` is available only when the length index is `n+1` for some `n`. While parameters are typically written as named arguments before the colon in a datatype declaration, and indices as arguments in a function type after the colon, Lean can infer when an argument after the colon is used as a parameter.

Indexed families allow the expression of complicated relationships between data, all checked by the compiler. The datatype's invariants can be encoded directly, and there is no way to violate them, not even temporarily. Informing the compiler about the datatype's invariants brings a major benefit: the compiler can now inform the programmer about what must be done to satisfy them. The strategic use of compile-time errors, especially those resulting from underscores, can make it possible to offload some of the programming thought process to Lean, freeing up the programmer's mind to worry about other things.

Encoding invariants using indexed families can lead to difficulties. First off, each invariant requires its own datatype, which then requires its own support libraries. `List.append` and `Vect.append` are not interchangeable, after all. This can lead to code duplication. Secondly, convenient use of indexed families requires that the recursive structure of functions used in types match the recursive structure of the programs being type checked. Programming with indexed families is the art of arranging for the right coincidences to occur. While it's possible to work around missing coincidences with appeals to equality proofs, it is difficult, and it leads to programs littered with cryptic justifications. Thirdly, running complicated code on large values during type checking can lead to compile-time slowdowns. Avoiding these slowdowns for complicated programs can require specialized techniques.

Definitional and Propositional Equality

Lean's type checker must, from time to time, check whether two types should be considered interchangeable. Because types can contain arbitrary programs, it must therefore be able to check arbitrary programs for equality. However, there is no efficient algorithm to check arbitrary programs for fully-general mathematical equality. To work around this, Lean contains two notions of equality:

- *Definitional equality* is an underapproximation of equality that essentially checks for equality of syntactic representation modulo computation and renaming of bound variables. Lean automatically checks for definitional equality in situations where it is required.

- *Propositional equality* must be explicitly proved and explicitly invoked by the programmer. In return, Lean automatically checks that the proofs are valid and that the invocations accomplish the right goal.

The two notions of equality represent a division of labor between programmers and Lean itself. Definitional equality is simple, but automatic, while propositional equality is manual, but expressive. Propositional equality can be used to unstick otherwise-stuck programs in types.

However, the frequent use of propositional equality to unstick type-level computation is typically a code smell. It typically means that coincidences were not well-engineered, and it's usually a better idea to either redesign the types and indices or to use a different technique to enforce the needed invariants. When propositional equality is instead used to prove that a program meets a specification, or as part of a subtype, there is less reason to be suspicious.

Interlude: Tactics, Induction, and Proofs

A Note on Proofs and User Interfaces

This book presents the process of writing proofs as if they are written in one go and submitted to Lean, which then replies with error messages that describe what remains to be done. The actual process of interacting with Lean is much more pleasant. Lean provides information about the proof as the cursor is moved through it and there are a number of interactive features that make proving easier. Please consult the documentation of your Lean development environment for more information.

The approach in this book that focuses on incrementally building a proof and showing the messages that result demonstrates the kinds of interactive feedback that Lean provides while writing a proof, even though it is much slower than the process used by experts. At the same time, seeing incomplete proofs evolve towards completeness is a useful perspective on proving. As your skill in writing proofs increases, Lean's feedback will come to feel less like errors and more like support for your own thought processes. Learning the interactive approach is very important.

Recursion and Induction

The functions `plusR_succ_left` and `plusR_zero_left` from the preceding chapter can be seen from two perspectives. On the one hand, they are recursive functions that build up evidence for a proposition, just as other recursive functions might construct a list, a string, or any other data structure. On the other, they also correspond to proofs by *mathematical induction*.

Mathematical induction is a proof technique where a statement is proven for *all* natural numbers in two steps:

1. The statement is shown to hold for 0. This is called the *base case*.
2. Under the assumption that the statement holds for some arbitrarily chosen number n, it is shown to hold for $n + 1$. This is called the *induction step*. The assumption that the statement holds for n is called the *induction hypothesis*.

Because it's impossible to check the statement for *every* natural number, induction provides a means of writing a proof that could, in principle, be expanded to any particular natural number. For example, if a concrete proof were desired for the number 3, then it could be constructed by using first the base case and then the induction step three times, to show the statement for 0, 1, 2, and finally 3. Thus, it proves the statement for all natural numbers.

The Induction Tactic

Writing proofs by induction as recursive functions that use helpers such as `congrArg` does not always do a good job of expressing the intentions behind the proof. While recursive functions indeed have the structure of induction, they should probably be viewed as an *encoding* of a proof. Furthermore, Lean's tactic system provides a number of opportunities to automate the construction of a proof that are not available when writing the recursive function explicitly. Lean provides an induction *tactic* that can carry out an entire proof by induction in a single tactic block. Behind the scenes, Lean constructs the recursive function that corresponds the use of induction.

To prove `plusR_zero_left` with the induction tactic, begin by writing its signature (using `theorem`, because this really is a proof). Then, use `by induction k` as the body of the definition:

```
theorem plusR_zero_left (k : Nat) : k = Nat.plusR 0 k := by
  induction k
```

The resulting message states that there are two goals:

```
unsolved goals
case zero
⊢ 0 = Nat.plusR 0 0

case succ
n✝ : Nat
a✝ : n✝ = Nat.plusR 0 n✝
⊢ n✝ + 1 = Nat.plusR 0 (n✝ + 1)
```

A tactic block is a program that is run while the Lean type checker processes a file, somewhat like a much more powerful C preprocessor macro. The tactics generate the actual program.

In the tactic language, there can be a number of goals. Each goal consists of a type together with some assumptions. These are analogous to using underscores as placeholders—the type in the goal represents what is to be proved, and the assumptions represent what is in-scope and can be used. In the case of the goal `case zero`, there are no assumptions and the type is `Nat.zero = Nat.plusR 0 Nat.zero`—this is the theorem statement with `0` instead of `k`. In the goal `case succ`, there are two assumptions, named `n✝` and `n_ih✝`. Behind the scenes, the `induction` tactic creates a dependent pattern match that refines the overall type, and `n✝` represents the argument to `Nat.succ` in the pattern. The assumption `n_ih✝` represents the result of calling the generated function recursively on `n✝`. Its type is the overall type of the theorem, just with `n✝` instead of `k`. The type to be fulfilled as part of the goal `case succ` is the overall theorem statement, with `Nat.succ n✝` instead of `k`.

The two goals that result from the use of the `induction` tactic correspond to the base case and the induction step in the description of mathematical induction. The base case is `case zero`. In `case succ`, `n_ih✝` corresponds to the induction hypothesis, while the whole of `case succ` is the induction step.

The next step in writing the proof is to focus on each of the two goals in turn. Just as `pure ()` can be used in a `do` block to indicate "do nothing", the tactic language has a statement `skip` that also does nothing. This can be used when Lean's syntax requires a tactic, but it's not yet clear which one should be used. Adding `with` to the end of the `induction` statement provides a syntax that is similar to pattern matching:

```
theorem plusR_zero_left (k : Nat) : k = Nat.plusR 0 k := by
  induction k with
  | zero => skip
  | succ n ih => skip
```

Each of the two `skip` statements has a message associated with it. The first shows the base case:

```
unsolved goals
case zero
⊢ 0 = Nat.plusR 0 0
```

The second shows the induction step:

```
unsolved goals
case succ
n : Nat
ih : n = Nat.plusR 0 n
⊢ n + 1 = Nat.plusR 0 (n + 1)
```

In the induction step, the inaccessible names with daggers have been replaced with the names provided after `succ`, namely `n` and `ih`.

The cases after `induction ... with` are not patterns: they consist of the name of a goal followed by zero or more names. The names are used for assumptions introduced in the goal; it is an error to provide more names than the goal introduces:

```
theorem plusR_zero_left (k : Nat) : k = Nat.plusR 0 k := by
  induction k with
  | zero => skip
  | succ n ih lots of names => skip
```

```
too many variable names provided at alternative 'succ', #5 provided, but #2
expected
```

Focusing on the base case, the `rfl` tactic works just as well inside of the `induction` tactic as it does in a recursive function:

```
theorem plusR_zero_left (k : Nat) : k = Nat.plusR 0 k := by
  induction k with
  | zero => rfl
  | succ n ih => skip
```

In the recursive function version of the proof, a type annotation made the expected type something that was easier to understand. In the tactic language, there are a number of specific ways to transform a goal to make it easier to solve. The `unfold` tactic replaces a defined name with its definition:

```
theorem plusR_zero_left (k : Nat) : k = Nat.plusR 0 k := by
  induction k with
  | zero => rfl
  | succ n ih =>
    unfold Nat.plusR
```

Now, the right-hand side of the equality in the goal has become `Nat.plusR 0 n + 1` instead of `Nat.plusR 0 (Nat.succ n)`:

```
unsolved goals
case succ
n : Nat
ih : n = Nat.plusR 0 n
⊢ n + 1 = Nat.plusR 0 n + 1
```

Instead of appealing to functions like `congrArg` and operators like `▸`, there are tactics that allow equality proofs to be used to transform proof goals. One of the most important is `rw`, which takes a list of equality proofs and replaces the left side with the right side in the goal. This almost does the right thing in `plusR_zero_left`:

```
theorem plusR_zero_left (k : Nat) : k = Nat.plusR 0 k := by
  induction k with
  | zero => rfl
  | succ n ih =>
    unfold Nat.plusR
    rw [ih]
```

However, the direction of the rewrite was incorrect. Replacing `n` with `Nat.plusR 0 n` made the goal more complicated rather than less complicated:

```
unsolved goals
case succ
n : Nat
ih : n = Nat.plusR 0 n
⊢ Nat.plusR 0 n + 1 = Nat.plusR 0 (Nat.plusR 0 n) + 1
```

This can be remedied by placing a left arrow before `ih` in the call to `rewrite`, which instructs it to replace the right-hand side of the equality with the left-hand side:

```
theorem plusR_zero_left (k : Nat) : k = Nat.plusR 0 k := by
  induction k with
  | zero => rfl
  | succ n ih =>
    unfold Nat.plusR
    rw [←ih]
```

This rewrite makes both sides of the equation identical, and Lean takes care of the `rfl` on its own. The proof is complete.

Tactic Golf

So far, the tactic language has not shown its true value. The above proof is no shorter than the recursive function; it's merely written in a domain-specific language instead of the full Lean language. But proofs with tactics can be shorter, easier, and more maintainable. Just as a lower score is better in the game of golf, a shorter proof is better in the game of tactic golf.

The induction step of `plusR_zero_left` can be proved using the simplification tactic `simp`. Using `simp` on its own does not help:

```
theorem plusR_zero_left (k : Nat) : k = Nat.plusR 0 k := by
  induction k with
  | zero => rfl
  | succ n ih =>
    simp
```

```
simp made no progress
```

However, `simp` can be configured to make use of a set of definitions. Just like `rw`, these arguments are provided in a list. Asking `simp` to take the definition of `Nat.plusR` into account leads to a simpler goal:

```
theorem plusR_zero_left (k : Nat) : k = Nat.plusR 0 k := by
  induction k with
  | zero => rfl
  | succ n ih =>
    simp [Nat.plusR]
```

```
unsolved goals
case succ
n : Nat
ih : n = Nat.plusR 0 n
⊢ n = Nat.plusR 0 n
```

In particular, the goal is now identical to the induction hypothesis. In addition to automatically proving simple equality statements, the simplifier automatically replaces goals like `Nat.succ A = Nat.succ B` with `A = B`. Because the induction hypothesis `ih` has exactly the right type, the `exact` tactic can indicate that it should be used:

```
theorem plusR_zero_left (k : Nat) : k = Nat.plusR 0 k := by
  induction k with
  | zero => rfl
  | succ n ih =>
    simp [Nat.plusR]
    exact ih
```

However, the use of `exact` is somewhat fragile. Renaming the induction hypothesis, which may happen while "golfing" the proof, would cause this proof to stop working. The `assumption` tactic solves the current goal if *any* of the assumptions match it:

```
theorem plusR_zero_left (k : Nat) : k = Nat.plusR 0 k := by
  induction k with
  | zero => rfl
  | succ n ih =>
    simp [Nat.plusR]
    assumption
```

This proof is no shorter than the prior proof that used unfolding and explicit rewriting. However, a series of transformations can make it much shorter, taking advantage of the fact that `simp` can solve many kinds of goals. The first step is to drop the `with` at the end of `induction`. For structured, readable proofs, the `with` syntax is convenient. It complains if any cases are missing, and it shows the structure of the induction clearly. But shortening proofs can often require a more liberal approach.

Using `induction` without `with` simply results in a proof state with two goals. The `case` tactic can be used to select one of them, just as in the branches of the `induction ... with` tactic. In other words, the following proof is equivalent to the prior proof:

```
theorem plusR_zero_left (k : Nat) : k = Nat.plusR 0 k := by
  induction k
  case zero => rfl
  case succ n ih =>
    simp [Nat.plusR]
    assumption
```

In a context with a single goal (namely, `k = Nat.plusR 0 k`), the `induction k` tactic yields two goals. In general, a tactic will either fail with an error or take a goal and transform it into zero or more new goals. Each new goal represents what remains to be proved. If the result is zero goals, then the tactic was a success, and that part of the proof is done.

The `<;>` operator takes two tactics as arguments, resulting in a new tactic. `T1 <;> T2` applies `T1` to the current goal, and then applies `T2` in *all* goals created by `T1`. In other words, `<;>` enables a general tactic that can solve many kinds of goals to be used on multiple new goals all at once. One such general tactic is `simp`.

Because `simp` can both complete the proof of the base case and make progress on the proof of the induction step, using it with `induction` and `<;>` shortens the proof:

```
theorem plusR_zero_left (k : Nat) : k = Nat.plusR 0 k := by
  induction k <;> simp [Nat.plusR]
```

This results in only a single goal, the transformed induction step:

```
unsolved goals
case succ
n✝ : Nat
a✝ : n✝ = Nat.plusR 0 n✝
⊢ n✝ = Nat.plusR 0 n✝
```

Running `assumption` in this goal completes the proof:

```
theorem plusR_zero_left (k : Nat) : k = Nat.plusR 0 k := by
  induction k <;> simp [Nat.plusR] <;> assumption
```

Here, `exact` would not have been possible, because `ih` was never explicitly named.

For beginners, this proof is not easier to read. However, a common pattern for expert users is to take care of a number of simple cases with powerful tactics like `simp`, allowing them to focus the text of the proof on the interesting cases. Additionally, these proofs tend to be more robust in the face of small changes to the functions and datatypes involved in the proof. The game of tactic golf is a useful part of developing good taste and style when writing proofs.

Induction on Other Datatypes

Mathematical induction proves a statement for natural numbers by providing a base case for `Nat.zero` and an induction step for `Nat.succ`. The principle of induction is also valid for other datatypes. Constructors without recursive arguments form the base cases, while constructors with recursive arguments form the induction steps. The ability to carry out proofs by induction is the very reason why they are called *inductive* datatypes.

One example of this is induction on binary trees. Induction on binary trees is a proof technique where a statement is proven for *all* binary trees in two steps:

1. The statement is shown to hold for `BinTree.leaf`. This is called the base case.
2. Under the assumption that the statement holds for some arbitrarily chosen trees `l` and `r`, it is shown to hold for `BinTree.branch l x r`, where `x` is an arbitrarily-chosen new data point. This is called the *induction step*. The assumptions that the statement holds for `l` and `r` are called the *induction hypotheses*.

`BinTree.count` counts the number of branches in a tree:

```
def BinTree.count : BinTree α → Nat
  | .leaf => 0
  | .branch l _ r =>
    1 + l.count + r.count
```

Mirroring a tree does not change the number of branches in it. This can be proven using induction on trees. The first step is to state the theorem and invoke `induction`:

```
theorem BinTree.mirror_count (t : BinTree α) : t.mirror.count = t.count := by
  induction t with
  | leaf => skip
  | branch l x r ihl ihr => skip
```

The base case states that counting the mirror of a leaf is the same as counting the leaf:

```
unsolved goals
case leaf
α : Type
⊢ leaf.mirror.count = leaf.count
```

The induction step allows the assumption that mirroring the left and right subtrees won't affect their branch counts, and requests a proof that mirroring a branch with these subtrees also preserves the overall branch count:

```
unsolved goals
case branch
α : Type
l : BinTree α
x : α
r : BinTree α
ihl : l.mirror.count = l.count
ihr : r.mirror.count = r.count
⊢ (l.branch x r).mirror.count = (l.branch x r).count
```

The base case is true because mirroring `leaf` results in `leaf`, so the left and right sides are definitionally equal. This can be expressed by using `simp` with instructions to unfold `BinTree.mirror`:

```
theorem BinTree.mirror_count (t : BinTree α) : t.mirror.count = t.count := by
  induction t with
  | leaf => simp [BinTree.mirror]
  | branch l x r ihl ihr => skip
```

In the induction step, nothing in the goal immediately matches the induction hypotheses. Simplifying using the definitions of `BinTree.count` and `BinTree.mirror` reveals the relationship:

```
theorem BinTree.mirror_count (t : BinTree α) : t.mirror.count = t.count := by
  induction t with
  | leaf => simp [BinTree.mirror]
  | branch l x r ihl ihr =>
    simp [BinTree.mirror, BinTree.count]
```

```
unsolved goals
case branch
α : Type
l : BinTree α
x : α
r : BinTree α
ihl : l.mirror.count = l.count
ihr : r.mirror.count = r.count
⊢ 1 + r.mirror.count + l.mirror.count = 1 + l.count + r.count
```

Both induction hypotheses can be used to rewrite the left-hand side of the goal into something almost like the right-hand side:

```
theorem BinTree.mirror_count (t : BinTree α) : t.mirror.count = t.count := by
  induction t with
  | leaf => simp [BinTree.mirror]
  | branch l x r ihl ihr =>
    simp [BinTree.mirror, BinTree.count]
    rw [ihl, ihr]
```

```
unsolved goals
case branch
α : Type
l : BinTree α
x : α
r : BinTree α
ihl : l.mirror.count = l.count
ihr : r.mirror.count = r.count
⊢ 1 + r.count + l.count = 1 + l.count + r.count
```

The `simp` tactic can use additional arithmetic identities when passed the `+arith` option. It is enough to prove this goal, yielding:

```
theorem BinTree.mirror_count (t : BinTree α) : t.mirror.count = t.count := by
  induction t with
  | leaf => simp [BinTree.mirror]
  | branch l x r ihl ihr =>
    simp [BinTree.mirror, BinTree.count]
    rw [ihl, ihr]
    simp +arith
```

In addition to definitions to be unfolded, the simplifier can also be passed names of equality proofs to use as rewrites while it simplifies proof goals. `BinTree.mirror_count` can also be written:

```
theorem BinTree.mirror_count (t : BinTree α) : t.mirror.count = t.count := by
  induction t with
  | leaf => simp [BinTree.mirror]
  | branch l x r ihl ihr =>
    simp +arith [BinTree.mirror, BinTree.count, ihl, ihr]
```

As proofs grow more complicated, listing assumptions by hand can become tedious. Furthermore, manually writing assumption names can make it more difficult to re-use proof steps for multiple subgoals. The argument `*` to `simp` or `simp_arith` instructs them to use *all* assumptions while simplifying or solving the goal. In other words, the proof could also be written:

```
theorem BinTree.mirror_count (t : BinTree α) : t.mirror.count = t.count := by
  induction t with
  | leaf => simp [BinTree.mirror]
  | branch l x r ihl ihr =>
    simp +arith [BinTree.mirror, BinTree.count, *]
```

Because both branches are using the simplifier, the proof can be reduced to:

```
theorem BinTree.mirror_count (t : BinTree α) : t.mirror.count = t.count := by
  induction t <;> simp +arith [BinTree.mirror, BinTree.count, *]
```

Exercises

- Prove `plusR_succ_left` using the `induction ... with` tactic.
- Rewrite the proof of `plus_succ_left` to use `<;>` in a single line.
- Prove that appending lists is associative using induction on lists: `theorem List.append_assoc (xs ys zs : List α) : xs ++ (ys ++ zs) = (xs ++ ys) ++ zs`

Programming, Proving, and Performance

This chapter is about programming. Programs need to compute the correct result, but they also need to do so efficiently. To write efficient functional programs, it's important to know both how to use data structures appropriately and how to think about the time and space needed to run a program.

This chapter is also about proofs. One of the most important data structures for efficient programming in Lean is the array, but safe use of arrays requires proving that array indices are in bounds. Furthermore, most interesting algorithms on arrays do not follow the pattern of structural recursion—instead, they iterate over the array. While these algorithms terminate, Lean will not necessarily be able to automatically check this. Proofs can be used to demonstrate why a program terminates.

Rewriting programs to make them faster often results in code that is more difficult to understand. Proofs can also show that two programs always compute the same answers, even if they do so with different algorithms or implementation techniques. In this way, the slow, straightforward program can serve as a specification for the fast, complicated version.

Combining proofs and programming allows programs to be both safe and efficient. Proofs allow elision of run-time bounds checks, they render many tests unnecessary, and they provide an extremely high level of confidence in a program without introducing any runtime performance overhead. However, proving theorems about programs can be time consuming and expensive, so other tools are often more economical.

Interactive theorem proving is a deep topic. This chapter provides only a taste, oriented towards the proofs that come up in practice while programming in Lean. Most interesting theorems are not closely related to programming. Please refer to Next Steps for a list of resources for learning more. Just as when learning programming, however, there's no substitute for hands-on experience when learning to write proofs—it's time to get started!

Tail Recursion

While Lean's `do`-notation makes it possible to use traditional loop syntax such as `for` and `while`, these constructs are translated behind the scenes to invocations of recursive functions. In most programming languages, recursive functions have a key disadvantage with respect to loops: loops consume no space on the stack, while recursive functions consume stack space proportional to the number of recursive calls. Stack space is typically limited, and it is often necessary to take algorithms that are naturally expressed as recursive functions and rewrite them as loops paired with an explicit mutable heap-allocated stack.

In functional programming, the opposite is typically true. Programs that are naturally expressed as mutable loops may consume stack space, while rewriting them to recursive functions can cause them to run quickly. This is due to a key aspect of functional programming languages: *tail-call elimination*. A tail call is a call from one function to another that can be compiled to an ordinary jump, replacing the current stack frame rather than pushing a new one, and tail-call elimination is the process of implementing this transformation.

Tail-call elimination is not just merely an optional optimization. Its presence is a fundamental part of being able to write efficient functional code. For it to be useful, it must be reliable. Programmers must be able to reliably identify tail calls, and they must be able to trust that the compiler will eliminate them.

The function `NonTail.sum` adds the contents of a list of `Nat`s:

```
def NonTail.sum : List Nat → Nat
  | [] => 0
  | x :: xs => x + sum xs
```

Applying this function to the list `[1, 2, 3]` results in the following sequence of evaluation steps:

```
NonTail.sum [1, 2, 3]
===>
1 + (NonTail.sum [2, 3])
===>
1 + (2 + (NonTail.sum [3]))
===>
1 + (2 + (3 + (NonTail.sum [])))
===>
1 + (2 + (3 + 0))
===>
1 + (2 + 3)
===>
1 + 5
===>
6
```

In the evaluation steps, parentheses indicate recursive calls to `NonTail.sum`. In other words, to add the three numbers, the program must first check that the list is non-empty. To add the head of the list (`1`) to the sum of the tail of the list, it is first necessary to compute the sum of the tail of the list:

```
1 + (NonTail.sum [2, 3])
```

But to compute the sum of the tail of the list, the program must check whether it is empty. It is not - the tail is itself a list with `2` at its head. The resulting step is waiting for the return of `NonTail.sum [3]`:

```
1 + (2 + (NonTail.sum [3]))
```

The whole point of the run-time call stack is to keep track of the values `1`, `2`, and `3` along with the instruction to add them to the result of the recursive call. As recursive calls are completed, control returns to the stack frame that made the call, so each step of addition is performed. Storing the heads of the list and the instructions to add them is not free; it takes space proportional to the length of the list.

The function `Tail.sum` also adds the contents of a list of `Nat`s:

```
def Tail.sumHelper (soFar : Nat) : List Nat → Nat
  | [] => soFar
  | x :: xs => sumHelper (x + soFar) xs

def Tail.sum (xs : List Nat) : Nat :=
  Tail.sumHelper 0 xs
```

Applying it to the list `[1, 2, 3]` results in the following sequence of evaluation steps:

```
Tail.sum [1, 2, 3]
===>
Tail.sumHelper 0 [1, 2, 3]
===>
Tail.sumHelper (0 + 1) [2, 3]
===>
Tail.sumHelper 1 [2, 3]
===>
Tail.sumHelper (1 + 2) [3]
===>
Tail.sumHelper 3 [3]
===>
Tail.sumHelper (3 + 3) []
===>
Tail.sumHelper 6 []
===>
6
```

The internal helper function calls itself recursively, but it does so in a way where nothing needs to be remembered in order to compute the final result. When `Tail.sumHelper` reaches its base case, control can be returned directly to `Tail.sum`, because the intermediate invocations of `Tail.sumHelper` simply return the results of their recursive calls unmodified. In other words, a single stack frame can be re-used for each recursive invocation of `Tail.sumHelper`. Tail-call elimination is exactly this re-use of the stack frame, and `Tail.sumHelper` is referred to as a *tail-recursive function*.

The first argument to `Tail.sumHelper` contains all of the information that would otherwise need to be tracked in the call stack—namely, the sum of the numbers encountered so far. In each recursive call, this argument is updated with new information, rather than adding new information to the call stack. Arguments like `soFar` that replace the information from the call stack are called *accumulators*.

At the time of writing and on the author's computer, `NonTail.sum` crashes with a stack overflow when passed a list with 216,856 or more entries. `Tail.sum`, on the other hand, can sum a list of 100,000,000 elements without a stack overflow. Because no new stack frames need to be pushed while running `Tail.sum`, it is completely equivalent to a `while` loop with a mutable variable that holds the current list. At each recursive call, the function argument on the stack is simply replaced with the next node of the list.

Tail and Non-Tail Positions

The reason why `Tail.sumHelper` is tail recursive is that the recursive call is in *tail position*. Informally speaking, a function call is in tail position when the caller does not need to modify the returned value in any way, but will just return it directly. More formally, tail position can be defined explicitly for expressions.

If a `match`-expression is in tail position, then each of its branches is also in tail position. Once a `match` has selected a branch, control proceeds immediately to it. Similarly, both branches of an `if`-expression are in tail position if the `if`-expression itself is in tail position. Finally, if a `let`-expression is in tail position, then its body is as well.

All other positions are not in tail position. The arguments to a function or a constructor are not in tail position because evaluation must track the function or constructor that will be applied to the argument's value. The body of an inner function is not in tail position because control may not even pass to it: function bodies are not evaluated until the function is called. Similarly, the body of a function type is not in tail position. To evaluate `E` in `(x : α) → E`, it is necessary to track that the resulting type must have `(x : α) → ...` wrapped around it.

In `NonTail.sum`, the recursive call is not in tail position because it is an argument to `+`. In `Tail.sumHelper`, the recursive call is in tail position because it is immediately underneath a pattern match, which itself is the body of the function.

At the time of writing, Lean only eliminates direct tail calls in recursive functions. This means that tail calls to `f` in `f`'s definition will be eliminated, but not tail calls to some other function `g`. While it is certainly possible to eliminate a tail call to some other function, saving a stack frame, this is not yet implemented in Lean.

Reversing Lists

The function `NonTail.reverse` reverses lists by appending the head of each sub-list to the end of the result:

```
def NonTail.reverse : List α → List α
  | [] => []
  | x :: xs => reverse xs ++ [x]
```

Using it to reverse `[1, 2, 3]` yields the following sequence of steps:

```
NonTail.reverse [1, 2, 3]
===>
(NonTail.reverse [2, 3]) ++ [1]
===>
((NonTail.reverse [3]) ++ [2]) ++ [1]
===>
(((NonTail.reverse []) ++ [3]) ++ [2]) ++ [1]
===>
(([] ++ [3]) ++ [2]) ++ [1]
===>
([3] ++ [2]) ++ [1]
===>
[3, 2] ++ [1]
===>
[3, 2, 1]
```

The tail-recursive version uses `x :: ·` instead of `· ++ [x]` on the accumulator at each step:

```
def Tail.reverseHelper (soFar : List α) : List α → List α
  | [] => soFar
  | x :: xs => reverseHelper (x :: soFar) xs

def Tail.reverse (xs : List α) : List α :=
  Tail.reverseHelper [] xs
```

This is because the context saved in each stack frame while computing with `NonTail.reverse` is applied beginning at the base case. Each "remembered" piece of context is executed in last-in, first-out order. On the other hand, the accumulator-passing version modifies the accumulator beginning from the first entry in the list, rather than the original base case, as can be seen in the series of reduction steps:

```
Tail.reverse [1, 2, 3]
===>
Tail.reverseHelper [] [1, 2, 3]
===>
Tail.reverseHelper [1] [2, 3]
===>
Tail.reverseHelper [2, 1] [3]
===>
Tail.reverseHelper [3, 2, 1] []
===>
[3, 2, 1]
```

In other words, the non-tail-recursive version starts at the base case, modifying the result of recursion from right to left through the list. The entries in the list affect the accumulator in a first-in, first-out order. The tail-recursive version with the accumulator starts at the head of the list, modifying an initial accumulator value from left to right through the list.

Because addition is commutative, nothing needed to be done to account for this in `Tail.sum`. Appending lists is not commutative, so care must be taken to find an operation that has the same effect when run in the opposite direction. Appending `[x]` after the result of the recursion in `NonTail.reverse` is analogous to adding `x` to the beginning of the list when the result is built in the opposite order.

Multiple Recursive Calls

In the definition of `BinTree.mirror`, there are two recursive calls:

```
def BinTree.mirror : BinTree α → BinTree α
  | .leaf => .leaf
  | .branch l x r => .branch (mirror r) x (mirror l)
```

Just as imperative languages would typically use a while loop for functions like `reverse` and `sum`, they would typically use recursive functions for this kind of traversal. This function cannot be straightforwardly rewritten to be tail recursive using accumulator-passing style, at least not using the techniques presented in this book.

Typically, if more than one recursive call is required for each recursive step, then it will be difficult to use accumulator-passing style. This difficulty is similar to the difficulty of rewriting a recursive function to use a loop and an explicit data structure, with the added complication of convincing Lean that the function terminates. However, as in `BinTree.mirror`, multiple recursive calls often indicate a data structure that has a constructor with multiple recursive occurrences of itself. In these cases, the depth of the structure is often logarithmic with respect to its overall size, which makes the tradeoff between stack and heap less stark. There are systematic techniques for making these functions tail-recursive, such as using *continuation-passing style* and *defunctionalization*, but they are outside the scope of this book.

Exercises

Translate each of the following non-tail-recursive functions into accumulator-passing tail-recursive functions:

```
def NonTail.length : List α → Nat
  | [] => 0
  | _ :: xs => NonTail.length xs + 1
```

```
def NonTail.factorial : Nat → Nat
  | 0 => 1
  | n + 1 => factorial n * (n + 1)
```

The translation of `NonTail.filter` should result in a program that takes constant stack space through tail recursion, and time linear in the length of the input list. A constant factor overhead is acceptable relative to the original:

```
def NonTail.filter (p : α → Bool) : List α → List α
  | [] => []
  | x :: xs =>
    if p x then
      x :: filter p xs
    else
      filter p xs
```

Proving Equivalence

Programs that have been rewritten to use tail recursion and an accumulator can look quite different from the original program. The original recursive function is often much easier to understand, but it runs the risk of exhausting the stack at run time. After testing both versions of the program on examples to rule out simple bugs, proofs can be used to show once and for all that the programs are equivalent.

Proving `sum` Equal

To prove that both versions of `sum` are equal, begin by writing the theorem statement with a stub proof:

```
theorem non_tail_sum_eq_tail_sum : NonTail.sum = Tail.sum := by
  skip
```

As expected, Lean describes an unsolved goal:

```
unsolved goals
⊢ NonTail.sum = Tail.sum
```

The `rfl` tactic cannot be applied here, because `NonTail.sum` and `Tail.sum` are not definitionally equal. Functions can be equal in more ways than just definitional equality, however. It is also possible to prove that two functions are equal by proving that they produce equal outputs for the same input. In other words, $f = g$ can be proved by proving that $f(x) = g(x)$ for all possible inputs x. This principle is called *function extensionality*. Function extensionality is exactly the reason why `NonTail.sum` equals `Tail.sum`: they both sum lists of numbers.

In Lean's tactic language, function extensionality is invoked using `funext`, followed by a name to be used for the arbitrary argument. The arbitrary argument is added as an assumption to the context, and the goal changes to require a proof that the functions applied to this argument are equal:

```
theorem non_tail_sum_eq_tail_sum : NonTail.sum = Tail.sum := by
  funext xs
```

```
unsolved goals
case h
xs : List Nat
⊢ NonTail.sum xs = Tail.sum xs
```

This goal can be proved by induction on the argument `xs`. Both `sum` functions return `0` when applied to the empty list, which serves as a base case. Adding a number to the beginning of the input list causes both functions to add that number to the result, which serves as an induction step. Invoking the `induction` tactic results in two goals:

```
theorem non_tail_sum_eq_tail_sum : NonTail.sum = Tail.sum := by
  funext xs
  induction xs with
  | nil => skip
  | cons y ys ih => skip
```

```
unsolved goals
case h.nil
⊢ NonTail.sum [] = Tail.sum []
```

```
unsolved goals
case h.cons
y : Nat
ys : List Nat
ih : NonTail.sum ys = Tail.sum ys
⊢ NonTail.sum (y :: ys) = Tail.sum (y :: ys)
```

The base case for `nil` can be solved using `rfl`, because both functions return `0` when passed the empty list:

```
theorem non_tail_sum_eq_tail_sum : NonTail.sum = Tail.sum := by
  funext xs
  induction xs with
  | nil => rfl
  | cons y ys ih => skip
```

The first step in solving the induction step is to simplify the goal, asking `simp` to unfold `NonTail.sum` and `Tail.sum`:

```
theorem non_tail_sum_eq_tail_sum : NonTail.sum = Tail.sum := by
  funext xs
  induction xs with
  | nil => rfl
  | cons y ys ih =>
    simp [NonTail.sum, Tail.sum]
```

```
unsolved goals
case h.cons
y : Nat
ys : List Nat
ih : NonTail.sum ys = Tail.sum ys
⊢ y + NonTail.sum ys = Tail.sumHelper 0 (y :: ys)
```

Unfolding `Tail.sum` revealed that it immediately delegates to `Tail.sumHelper`, which should also be simplified:

```
theorem non_tail_sum_eq_tail_sum : NonTail.sum = Tail.sum := by
  funext xs
  induction xs with
  | nil => rfl
  | cons y ys ih =>
    simp [NonTail.sum, Tail.sum, Tail.sumHelper]
```

In the resulting goal, `sumHelper` has taken a step of computation and added `y` to the accumulator:

```
unsolved goals
case h.cons
y : Nat
ys : List Nat
ih : NonTail.sum ys = Tail.sum ys
⊢ y + NonTail.sum ys = Tail.sumHelper y ys
```

Rewriting with the induction hypothesis removes all mentions of `NonTail.sum` from the goal:

```
theorem non_tail_sum_eq_tail_sum : NonTail.sum = Tail.sum := by
  funext xs
  induction xs with
  | nil => rfl
  | cons y ys ih =>
    simp [NonTail.sum, Tail.sum, Tail.sumHelper]
    rw [ih]
```

```
unsolved goals
case h.cons
y : Nat
ys : List Nat
ih : NonTail.sum ys = Tail.sum ys
⊢ y + Tail.sum ys = Tail.sumHelper y ys
```

This new goal states that adding some number to the sum of a list is the same as using that number as the initial accumulator in `sumHelper`. For the sake of clarity, this new goal can be proved as a separate theorem:

```
theorem helper_add_sum_accum (xs : List Nat) (n : Nat) :
    n + Tail.sum xs = Tail.sumHelper n xs := by
  skip
```

```
unsolved goals
xs : List Nat
n : Nat
⊢ n + Tail.sum xs = Tail.sumHelper n xs
```

Once again, this is a proof by induction where the base case uses `rfl`:

```
theorem helper_add_sum_accum (xs : List Nat) (n : Nat) :
    n + Tail.sum xs = Tail.sumHelper n xs := by
  induction xs with
  | nil => rfl
  | cons y ys ih => skip
```

```
unsolved goals
case cons
n y : Nat
ys : List Nat
ih : n + Tail.sum ys = Tail.sumHelper n ys
⊢ n + Tail.sum (y :: ys) = Tail.sumHelper n (y :: ys)
```

Because this is an inductive step, the goal should be simplified until it matches the induction hypothesis `ih`. Simplifying, using the definitions of `Tail.sum` and `Tail.sumHelper`, results in the following:

```
theorem helper_add_sum_accum (xs : List Nat) (n : Nat) :
    n + Tail.sum xs = Tail.sumHelper n xs := by
  induction xs with
  | nil => rfl
  | cons y ys ih =>
    simp [Tail.sum, Tail.sumHelper]
```

```
unsolved goals
case cons
n y : Nat
ys : List Nat
ih : n + Tail.sum ys = Tail.sumHelper n ys
⊢ n + Tail.sumHelper y ys = Tail.sumHelper (y + n) ys
```

Ideally, the induction hypothesis could be used to replace `Tail.sumHelper (y + n) ys`, but they don't match. The induction hypothesis can be used for `Tail.sumHelper n ys`, not `Tail.sumHelper (y + n) ys`. In other words, this proof is stuck.

A Second Attempt

Rather than attempting to muddle through the proof, it's time to take a step back and think. Why is it that the tail-recursive version of the function is equal to the non-tail-recursive version? Fundamentally speaking, at each entry in the list, the accumulator grows by the same amount as would be added to the result of the recursion. This insight can be used to write an elegant proof. Crucially, the proof by induction must be set up such that the induction hypothesis can be applied to *any* accumulator value.

Discarding the prior attempt, the insight can be encoded as the following statement:

```
theorem non_tail_sum_eq_helper_accum (xs : List Nat) :
    (n : Nat) → n + NonTail.sum xs = Tail.sumHelper n xs := by
  skip
```

In this statement, it's very important that `n` is part of the type that's after the colon. The resulting goal begins with `∀ (n : Nat)`, which is short for "For all `n`":

```
unsolved goals
xs : List Nat
⊢ ∀ (n : Nat), n + NonTail.sum xs = Tail.sumHelper n xs
```

Using the induction tactic results in goals that include this "for all" statement:

```
theorem non_tail_sum_eq_helper_accum (xs : List Nat) :
    (n : Nat) → n + NonTail.sum xs = Tail.sumHelper n xs := by
  induction xs with
  | nil => skip
  | cons y ys ih => skip
```

In the `nil` case, the goal is:

```
unsolved goals
case nil
⊢ ∀ (n : Nat), n + NonTail.sum [] = Tail.sumHelper n []
```

For the induction step for `cons`, both the induction hypothesis and the specific goal contain the "for all `n`":

```
unsolved goals
case cons
y : Nat
ys : List Nat
ih : ∀ (n : Nat), n + NonTail.sum ys = Tail.sumHelper n ys
⊢ ∀ (n : Nat), n + NonTail.sum (y :: ys) = Tail.sumHelper n (y :: ys)
```

In other words, the goal has become more challenging to prove, but the induction hypothesis is correspondingly more useful.

A mathematical proof for a statement that beings with "for all x" should assume some arbitrary x, and prove the statement. "Arbitrary" means that no additional properties of x are assumed, so the resulting statement will work for *any* x. In Lean, a "for all" statement is a dependent function: no matter which specific value it is applied to, it will return evidence of the proposition. Similarly, the process of picking an arbitrary x is the same as using `fun x => ...`. In the tactic language, this process of selecting an arbitrary x is performed using the `intro` tactic, which produces the function behind the scenes when the tactic script has completed. The `intro` tactic should be provided with the name to be used for this arbitrary value.

Using the `intro` tactic in the `nil` case removes the `∀ (n : Nat),` from the goal, and adds an assumption `n : Nat`:

```
theorem non_tail_sum_eq_helper_accum (xs : List Nat) :
    (n : Nat) → n + NonTail.sum xs = Tail.sumHelper n xs := by
  induction xs with
  | nil => intro n
  | cons y ys ih => skip
```

```
unsolved goals
case nil
n : Nat
⊢ n + NonTail.sum [] = Tail.sumHelper n []
```

Both sides of this propositional equality are definitionally equal to `n`, so `rfl` suffices:

```
theorem non_tail_sum_eq_helper_accum (xs : List Nat) :
    (n : Nat) → n + NonTail.sum xs = Tail.sumHelper n xs := by
  induction xs with
  | nil =>
    intro n
    rfl
  | cons y ys ih => skip
```

The `cons` goal also contains a "for all":

```
unsolved goals
case cons
y : Nat
ys : List Nat
ih : ∀ (n : Nat), n + NonTail.sum ys = Tail.sumHelper n ys
⊢ ∀ (n : Nat), n + NonTail.sum (y :: ys) = Tail.sumHelper n (y :: ys)
```

This suggests the use of `intro`.

```
theorem non_tail_sum_eq_helper_accum (xs : List Nat) :
    (n : Nat) → n + NonTail.sum xs = Tail.sumHelper n xs := by
  induction xs with
  | nil =>
    intro n
    rfl
  | cons y ys ih =>
    intro n
```

```
unsolved goals
case cons
y : Nat
ys : List Nat
ih : ∀ (n : Nat), n + NonTail.sum ys = Tail.sumHelper n ys
n : Nat
⊢ n + NonTail.sum (y :: ys) = Tail.sumHelper n (y :: ys)
```

The proof goal now contains both `NonTail.sum` and `Tail.sumHelper` applied to `y :: ys`. The simplifier can make the next step more clear:

```
theorem non_tail_sum_eq_helper_accum (xs : List Nat) :
    (n : Nat) → n + NonTail.sum xs = Tail.sumHelper n xs := by
  induction xs with
  | nil =>
    intro n
    rfl
  | cons y ys ih =>
    intro n
    simp [NonTail.sum, Tail.sumHelper]
```

```
unsolved goals
case cons
y : Nat
ys : List Nat
ih : ∀ (n : Nat), n + NonTail.sum ys = Tail.sumHelper n ys
n : Nat
⊢ n + (y + NonTail.sum ys) = Tail.sumHelper (y + n) ys
```

This goal is very close to matching the induction hypothesis. There are two ways in which it does not match:

- The left-hand side of the equation is `n + (y + NonTail.sum ys)`, but the induction hypothesis needs the left-hand side to be a number added to `NonTail.sum ys`. In other words, this goal should be rewritten to `(n + y) + NonTail.sum ys`, which is valid because addition of natural numbers is associative.

- When the left side has been rewritten to `(y + n) + NonTail.sum ys`, the accumulator argument on the right side should be `n + y` rather than `y + n` in order to match. This rewrite is valid because addition is also commutative.

The associativity and commutativity of addition have already been proved in Lean's standard library. The proof of associativity is named `Nat.add_assoc`, and its type is `(n m k : Nat) → (n + m) + k = n + (m + k)`, while the proof of commutativity is called `Nat.add_comm` and has type `(n m : Nat) → n + m = m + n`. Normally, the `rw` tactic is provided with an expression whose type is an equality. However, if the argument is instead a dependent function whose return type is an equality, it attempts to find arguments to the function that would allow the equality to match something in the goal. There is only one opportunity to apply associativity, though the direction of the rewrite must be reversed because the right side of the equality in `Nat.add_assoc` is the one that matches the proof goal:

```
theorem non_tail_sum_eq_helper_accum (xs : List Nat) :
    (n : Nat) → n + NonTail.sum xs = Tail.sumHelper n xs := by
  induction xs with
  | nil =>
    intro n
    rfl
  | cons y ys ih =>
    intro n
    simp [NonTail.sum, Tail.sumHelper]
    rw [←Nat.add_assoc]
```

```
unsolved goals
case cons
y : Nat
ys : List Nat
ih : ∀ (n : Nat), n + NonTail.sum ys = Tail.sumHelper n ys
n : Nat
⊢ n + y + NonTail.sum ys = Tail.sumHelper (y + n) ys
```

Rewriting directly with `Nat.add_comm`, however, leads to the wrong result. The `rw` tactic guesses the wrong location for the rewrite, leading to an unintended goal:

```
theorem non_tail_sum_eq_helper_accum (xs : List Nat) :
    (n : Nat) → n + NonTail.sum xs = Tail.sumHelper n xs := by
  induction xs with
  | nil =>
    intro n
    rfl
  | cons y ys ih =>
    intro n
    simp [NonTail.sum, Tail.sumHelper]
    rw [←Nat.add_assoc]
    rw [Nat.add_comm]
```

```
unsolved goals
case cons
y : Nat
ys : List Nat
ih : ∀ (n : Nat), n + NonTail.sum ys = Tail.sumHelper n ys
n : Nat
⊢ NonTail.sum ys + (n + y) = Tail.sumHelper (y + n) ys
```

This can be fixed by explicitly providing `y` and `n` as arguments to `Nat.add_comm`:

```
theorem non_tail_sum_eq_helper_accum (xs : List Nat) :
    (n : Nat) → n + NonTail.sum xs = Tail.sumHelper n xs := by
  induction xs with
  | nil =>
    intro n
    rfl
  | cons y ys ih =>
    intro n
    simp [NonTail.sum, Tail.sumHelper]
    rw [←Nat.add_assoc]
    rw [Nat.add_comm y n]
```

```
unsolved goals
case cons
y : Nat
ys : List Nat
ih : ∀ (n : Nat), n + NonTail.sum ys = Tail.sumHelper n ys
n : Nat
⊢ n + y + NonTail.sum ys = Tail.sumHelper (n + y) ys
```

The goal now matches the induction hypothesis. In particular, the induction hypothesis's type is a dependent function type. Applying `ih` to `n + y` results in exactly the desired type. The `exact` tactic completes a proof goal if its argument has exactly the desired type:

```
theorem non_tail_sum_eq_helper_accum (xs : List Nat) :
    (n : Nat) → n + NonTail.sum xs = Tail.sumHelper n xs := by
  induction xs with
  | nil => intro n; rfl
  | cons y ys ih =>
    intro n
    simp [NonTail.sum, Tail.sumHelper]
    rw [←Nat.add_assoc]
    rw [Nat.add_comm y n]
    exact ih (n + y)
```

The actual proof requires only a little additional work to get the goal to match the helper's type. The first step is still to invoke function extensionality:

```
theorem non_tail_sum_eq_tail_sum : NonTail.sum = Tail.sum := by
  funext xs
```

```
unsolved goals
case h
xs : List Nat
⊢ NonTail.sum xs = Tail.sum xs
```

The next step is unfold `Tail.sum`, exposing `Tail.sumHelper`:

```
theorem non_tail_sum_eq_tail_sum : NonTail.sum = Tail.sum := by
  funext xs
  simp [Tail.sum]
```

```
unsolved goals
case h
xs : List Nat
⊢ NonTail.sum xs = Tail.sumHelper 0 xs
```

Having done this, the types almost match. However, the helper has an additional addend on the left side. In other words, the proof goal is `NonTail.sum xs = Tail.sumHelper 0 xs`, but applying `non_tail_sum_eq_helper_accum` to `xs` and `0` yields the type `0 + NonTail.sum xs = Tail.sumHelper 0 xs`. Another standard library proof, `Nat.zero_add`, has type `(n : Nat) → 0 + n = n`. Applying this function to `NonTail.sum xs` results in an expression with type `0 + NonTail.sum xs = NonTail.sum xs`, so rewriting from right to left results in the desired goal:

```
theorem non_tail_sum_eq_tail_sum : NonTail.sum = Tail.sum := by
  funext xs
  simp [Tail.sum]
  rw [←Nat.zero_add (NonTail.sum xs)]
```

```
unsolved goals
case h
xs : List Nat
⊢ 0 + NonTail.sum xs = Tail.sumHelper 0 xs
```

Finally, the helper can be used to complete the proof:

```
theorem non_tail_sum_eq_tail_sum : NonTail.sum = Tail.sum := by
  funext xs
  simp [Tail.sum]
  rw [←Nat.zero_add (NonTail.sum xs)]
  exact non_tail_sum_eq_helper_accum xs 0
```

This proof demonstrates a general pattern that can be used when proving that an accumulator-passing tail-recursive function is equal to the non-tail-recursive version. The first step is to discover the relationship between the starting accumulator argument and the final result. For instance, beginning `Tail.sumHelper` with an accumulator of `n` results in the final sum being added to `n`, and beginning `Tail.reverseHelper` with an accumulator of `ys` results in the final reversed list being prepended to `ys`. The second step is to write down this relationship as a theorem statement and prove it by induction. While the accumulator is always initialized with some neutral value in practice, such as `0` or `[]`, this more general statement that allows the starting accumulator to be any value is what's needed to get a strong enough induction hypothesis. Finally, using this helper theorem with the actual initial accumulator value results in the desired proof. For example, in `non_tail_sum_eq_tail_sum`, the accumulator is specified to be `0`. This may require rewriting the goal to make the neutral initial accumulator values occur in the right place.

Exercise

Warming Up

Write your own proofs for `Nat.zero_add`, `Nat.add_assoc`, and `Nat.add_comm` using the `induction` tactic.

More Accumulator Proofs

Reversing Lists

Adapt the proof for `sum` into a proof for `NonTail.reverse` and `Tail.reverse`. The first step is to think about the relationship between the accumulator value being passed to `Tail.reverseHelper` and the non-tail-recursive reverse. Just as adding a number to the accumulator in `Tail.sumHelper` is the same as adding it to the overall sum, using `List.cons` to add a new entry to the accumulator in `Tail.reverseHelper` is equivalent to some change to the overall result. Try three or four different accumulator values with pencil and paper until the relationship becomes clear. Use this relationship to prove a suitable helper theorem. Then, write down the overall theorem. Because `NonTail.reverse` and `Tail.reverse` are polymorphic, stating their equality requires the use of `@` to stop Lean from trying to figure out which type to use for `α`. Once `α` is treated as an ordinary argument, `funext` should be invoked with both `α` and `xs`:

```
theorem non_tail_reverse_eq_tail_reverse : @NonTail.reverse = @Tail.reverse :=
by
  funext α xs
```

This results in a suitable goal:

```
unsolved goals
case h.h
α : Type u_1
xs : List α
⊢ NonTail.reverse xs = Tail.reverse xs
```

Factorial

Prove that `NonTail.factorial` from the exercises in the previous section is equal to your tail-recursive solution by finding the relationship between the accumulator and the result and proving a suitable helper theorem.

Arrays and Termination

To write efficient code, it is important to select appropriate data structures. Linked lists have their place: in some applications, the ability to share the tails of lists is very important. However, most use cases for a variable-length sequential collection of data are better served by arrays, which have both less memory overhead and better locality.

Arrays, however, have two drawbacks relative to lists:

1. Arrays are accessed through indexing, rather than by pattern matching, which imposes proof obligations in order to maintain safety.
2. A loop that processes an entire array from left to right is a tail-recursive function, but it does not have an argument that decreases on each call.

Making effective use of arrays requires knowing how to prove to Lean that an array index is in bounds, and how to prove that an array index that approaches the size of the array also causes the program to terminate. Both of these are expressed using an inequality proposition, rather than propositional equality.

Inequality

Because different types have different notions of ordering, inequality is governed by two type classes, called `LE` and `LT`. The table in the section on standard type classes describes how these classes relate to the syntax:

Expression	Desugaring	Class Name
`x < y`	`LT.lt x y`	`LT`
`x ≤ y`	`LE.le x y`	`LE`
`x > y`	`LT.lt y x`	`LT`
`x ≥ y`	`LE.le y x`	`LE`

In other words, a type may customize the meaning of the `<` and `≤` operators, while `>` and `≥` derive their meanings from `<` and `≤`. The classes `LT` and `LE` have methods that return propositions rather than `Bool`s:

```
class LE (α : Type u) where
  le : α → α → Prop

class LT (α : Type u) where
  lt : α → α → Prop
```

The instance of `LE` for `Nat` delegates to `Nat.le`:

```
instance : LE Nat where
  le := Nat.le
```

Defining `Nat.le` requires a feature of Lean that has not yet been presented: it is an inductively-defined relation.

Inductively-Defined Propositions, Predicates, and Relations

`Nat.le` is an *inductively-defined relation*. Just as `inductive` can be used to create new datatypes, it can be used to create new propositions. When a proposition takes an argument, it is referred to as a *predicate* that may be true for some, but not all, potential arguments. Propositions that take multiple arguments are called *relations*.

Each constructor of an inductively defined proposition is a way to prove it. In other words, the declaration of the proposition describes the different forms of evidence that it is true. A proposition with no arguments that has a single constructor can be quite easy to prove:

```
inductive EasyToProve : Prop where
  | heresTheProof : EasyToProve
```

The proof consists of using its constructor:

```
theorem fairlyEasy : EasyToProve := by
  constructor
```

In fact, the proposition `True`, which should always be easy to prove, is defined just like `EasyToProve`:

```
inductive True : Prop where
  | intro : True
```

Inductively-defined propositions that don't take arguments are not nearly as interesting as inductively-defined datatypes. This is because data is interesting in its own right—the natural number `3` is different from the number `35`, and someone who has ordered 3 pizzas will be upset if 35 arrive at their door 30 minutes later. The constructors of a proposition describe ways in which the proposition can be true, but once a proposition has been proved, there is no need to know *which* underlying constructors were used. This is why most interesting inductively-defined types in the `Prop` universe take arguments.

The inductively-defined predicate `IsThree` states that its argument is three:

```
inductive IsThree : Nat → Prop where
  | isThree : IsThree 3
```

The mechanism used here is just like indexed families such as `HasCol`, except the resulting type is a proposition that can be proved rather than data that can be used.

Using this predicate, it is possible to prove that three is indeed three:

```
theorem three_is_three : IsThree 3 := by
  constructor
```

Similarly, `IsFive` is a predicate that states that its argument is `5`:

```
inductive IsFive : Nat → Prop where
  | isFive : IsFive 5
```

If a number is three, then the result of adding two to it should be five. This can be expressed as a theorem statement:

```
theorem three_plus_two_five : IsThree n → IsFive (n + 2) := by
  skip
```

The resulting goal has a function type:

```
unsolved goals
n : Nat
⊢ IsThree n → IsFive (n + 2)
```

Thus, the `intro` tactic can be used to convert the argument into an assumption:

```
theorem three_plus_two_five : IsThree n → IsFive (n + 2) := by
  intro three
```

```
unsolved goals
n : Nat
three : IsThree n
⊢ IsFive (n + 2)
```

Given the assumption that `n` is three, it should be possible to use the constructor of

`IsFive` to complete the proof:

```
theorem three_plus_two_five : IsThree n → IsFive (n + 2) := by
  intro three
  constructor
```

However, this results in an error:

```
tactic 'constructor' failed, no applicable constructor found
n : Nat
three : IsThree n
⊢ IsFive (n + 2)
```

This error occurs because `n + 2` is not definitionally equal to `5`. In an ordinary function definition, dependent pattern matching on the assumption `three` could be used to refine `n` to `3`. The tactic equivalent of dependent pattern matching is `cases`, which has a syntax similar to that of `induction`:

```
theorem three_plus_two_five : IsThree n → IsFive (n + 2) := by
  intro three
  cases three with
  | isThree => skip
```

In the remaining case, `n` has been refined to `3`:

```
unsolved goals
case isThree
⊢ IsFive (3 + 2)
```

Because `3 + 2` is definitionally equal to `5`, the constructor is now applicable:

```
theorem three_plus_two_five : IsThree n → IsFive (n + 2) := by
  intro three
  cases three with
  | isThree => constructor
```

The standard false proposition `False` has no constructors, making it impossible to provide direct evidence for. The only way to provide evidence for `False` is if an assumption is itself impossible, similarly to how `nomatch` can be used to mark code that the type system can see is unreachable. As described in the initial Interlude on proofs, the negation `Not A` is short for `A → False`. `Not A` can also be written `¬A`.

It is not the case that four is three:

```
theorem four_is_not_three : ¬ IsThree 4 := by
  skip
```

The initial proof goal contains `Not`:

```
unsolved goals
⊢ ¬IsThree 4
```

The fact that it's actually a function type can be exposed using `unfold`:

```
theorem four_is_not_three : ¬ IsThree 4 := by
  unfold Not
```

```
unsolved goals
⊢ IsThree 4 → False
```

Because the goal is a function type, `intro` can be used to convert the argument into an assumption. There is no need to keep `unfold`, as `intro` can unfold the definition of `Not` itself:

```
theorem four_is_not_three : ¬ IsThree 4 := by
  intro h
```

```
unsolved goals
h : IsThree 4
⊢ False
```

In this proof, the `cases` tactic solves the goal immediately:

```
theorem four_is_not_three : ¬ IsThree 4 := by
  intro h
  cases h
```

Just as a pattern match on a `Vect String 2` doesn't need to include a case for `Vect.nil`, a proof by cases over `IsThree 4` doesn't need to include a case for `isThree`.

Inequality of Natural Numbers

The definition of `Nat.le` has a parameter and an index:

```
inductive Nat.le (n : Nat) : Nat → Prop
  | refl : Nat.le n n
  | step : Nat.le n m → Nat.le n (m + 1)
```

The parameter `n` is the number that should be smaller, while the index is the number that should be greater than or equal to `n`. The `refl` constructor is used when both numbers are equal, while the `step` constructor is used when the index is greater than `n`.

From the perspective of evidence, a proof that $n \leq k$ consists of finding some number d such that $n + d = m$. In Lean, the proof then consists of a `Nat.le.refl` constructor wrapped by d instances of `Nat.le.step`. Each `step` constructor adds one to its index argument, so d `step` constructors adds d to the larger number. For example, evidence that four is less than or equal to seven consists of three `step`s around a `refl`:

```
theorem four_le_seven : 4 ≤ 7 :=
  open Nat.le in
  step (step (step refl))
```

The strict less-than relation is defined by adding one to the number on the left:

```
def Nat.lt (n m : Nat) : Prop :=
  Nat.le (n + 1) m

instance : LT Nat where
  lt := Nat.lt
```

Evidence that four is strictly less than seven consists of two `step`'s around a `refl`:

```
theorem four_lt_seven : 4 < 7 :=
  open Nat.le in
  step (step refl)
```

This is because `4 < 7` is equivalent to `5 ≤ 7`.

Proving Termination

The function `Array.map` transforms an array with a function, returning a new array that contains the result of applying the function to each element of the input array. Writing it as a tail-recursive function follows the usual pattern of delegating to a function that passes the output array in an accumulator. The accumulator is initialized with an empty array. The accumulator-passing helper function also takes an argument that tracks the current index into the array, which starts at `0`:

```
def Array.map (f : α → β) (arr : Array α) : Array β :=
  arrayMapHelper f arr Array.empty 0
```

The helper should, at each iteration, check whether the index is still in bounds. If so, it should loop again with the transformed element added to the end of the accumulator and the index incremented by `1`. If not, then it should terminate and return the accumulator. An initial implementation of this code fails because Lean is unable to prove that the array index is valid:

```
def arrayMapHelper (f : α → β) (arr : Array α) (soFar : Array β) (i : Nat) :
Array β :=
  if i < arr.size then
    arrayMapHelper f arr (soFar.push (f arr[i])) (i + 1)
  else soFar
```

```
failed to prove index is valid, possible solutions:
  - Use `have`-expressions to prove the index is valid
  - Use `a[i]!` notation instead, runtime check is performed, and 'Panic' error
message is produced if index is not valid
  - Use `a[i]?` notation instead, result is an `Option` type
  - Use `a[i]'h` notation instead, where `h` is a proof that index is valid
α : Type ?u.1290
β : Type ?u.1293
f : α → β
arr : Array α
soFar : Array β
i : Nat
⊢ i < arr.size
```

However, the conditional expression already checks the precise condition that the array index's validity demands (namely, `i < arr.size`). Adding a name to the `if` resolves the issue, because it adds an assumption that the array indexing tactic can use:

```
def arrayMapHelper (f : α → β) (arr : Array α) (soFar : Array β) (i : Nat) :
Array β :=
  if inBounds : i < arr.size then
    arrayMapHelper f arr (soFar.push (f arr[i])) (i + 1)
  else soFar
```

Lean accepts the modified program, even though the recursive call is not made on an argument to one of the input constructors. In fact, both the accumulator and the index grow, rather than shrinking.

Behind the scenes, Lean's proof automation constructs a termination proof. Reconstructing this proof can make it easier to understand the cases that Lean cannot automatically recognize.

Why does `arrayMapHelper` terminate? Each iteration checks whether the index `i` is still in bounds for the array `arr`. If so, `i` is incremented and the loop repeats. If not, the program terminates. Because `arr.size` is a finite number, `i` can be incremented only a finite number of times. Even though no argument to the function decreases on each call, `arr.size - i` decreases toward zero.

The value that decreases at each recursive call is called a *measure*. Lean can be instructed to use a specific expression as the measure of termination by providing a `termination_by` clause at the end of a definition. For `arrayMapHelper`, the explicit measure looks like this:

```
def arrayMapHelper (f : α → β) (arr : Array α) (soFar : Array β) (i : Nat) :
Array β :=
  if inBounds : i < arr.size then
    arrayMapHelper f arr (soFar.push (f arr[i])) (i + 1)
  else soFar
termination_by arr.size - i
```

A similar termination proof can be used to write `Array.find`, a function that finds the first element in an array that satisfies a Boolean function and returns both the element and its index:

```
def Array.find (arr : Array α) (p : α → Bool) : Option (Nat × α) :=
  findHelper arr p 0
```

Once again, the helper function terminates because `arr.size - i` decreases as `i` increases:

```
def findHelper (arr : Array α) (p : α → Bool) (i : Nat) : Option (Nat × α) :=
  if h : i < arr.size then
    let x := arr[i]
    if p x then
      some (i, x)
    else findHelper arr p (i + 1)
  else none
```

Adding a question mark to `termination_by` (that is, using `termination_by?`) causes Lean to explicitly suggest the measure that it chose:

```
def findHelper (arr : Array α) (p : α → Bool) (i : Nat) : Option (Nat × α) :=
  if h : i < arr.size then
    let x := arr[i]
    if p x then
      some (i, x)
    else findHelper arr p (i + 1)
  else none
termination_by?
```

```
Try this: termination_by arr.size - i
```

Not all termination arguments are as quite as simple as this one. However, the basic structure of identifying some expression based on the function's arguments that will decrease in each call occurs in all termination proofs. Sometimes, creativity can be required in order to figure out just why a function terminates, and sometimes Lean requires additional proofs in order to accept that the measure in fact decreases.

Exercises

- Implement a `ForM (Array α)` instance on arrays using a tail-recursive accumulator-passing function and a `termination_by` clause.
- Reimplement `Array.map`, `Array.find`, and the `ForM` instance using `for ... in ...` loops in the identity monad and compare the resulting code.
- Reimplement array reversal using a `for ... in ...` loop in the identity monad. Compare it to the tail-recursive function.

More Inequalities

Lean's built-in proof automation is sufficient to check that `arrayMapHelper` and `findHelper` terminate. All that was needed was to provide an expression whose value decreases with each recursive call. However, Lean's built-in automation is not magic, and it often needs some help.

Merge Sort

One example of a function whose termination proof is non-trivial is merge sort on `List`. Merge sort consists of two phases: first, a list is split in half. Each half is sorted using merge sort, and then the results are merged using a function that combines two sorted lists into a larger sorted list. The base cases are the empty list and the singleton list, both of which are already considered to be sorted.

To merge two sorted lists, there are two basic cases to consider:

1. If one of the input lists is empty, then the result is the other list.
2. If both lists are non-empty, then their heads should be compared. The result of the function is the smaller of the two heads, followed by the result of merging the remaining entries of both lists.

This is not structurally recursive on either list. The recursion terminates because an entry is removed from one of the two lists in each recursive call, but it could be either list. Behind the scenes, Lean uses this fact to prove that it terminates:

```
def merge [Ord α] (xs : List α) (ys : List α) : List α :=
  match xs, ys with
  | [], _ => ys
  | _, [] => xs
  | x'::xs', y'::ys' =>
    match Ord.compare x' y' with
    | .lt | .eq => x' :: merge xs' (y' :: ys')
    | .gt => y' :: merge (x'::xs') ys'
```

A simple way to split a list is to add each entry in the input list to two alternating output lists:

```
def splitList (lst : List α) : (List α × List α) :=
  match lst with
  | [] => ([], [])
  | x :: xs =>
    let (a, b) := splitList xs
    (x :: b, a)
```

This splitting function is structurally recursive.

Merge sort checks whether a base case has been reached. If so, it returns the input list. If not, it splits the input, and merges the result of sorting each half:

```
def mergeSort [Ord α] (xs : List α) : List α :=
  if h : xs.length < 2 then
    match xs with
    | [] => []
    | [x] => [x]
  else
    let halves := splitList xs
    merge (mergeSort halves.fst) (mergeSort halves.snd)
```

Lean's pattern match compiler is able to tell that the assumption `h` introduced by the `if` that tests whether `xs.length < 2` rules out lists longer than one entry, so there is no "missing cases" error. However, even though this program always terminates, it is not structurally recursive, and Lean is unable to automatically discover a decreasing measure:

```
fail to show termination for
  mergeSort
with errors
failed to infer structural recursion:
Not considering parameter α of mergeSort:
  it is unchanged in the recursive calls
Not considering parameter #2 of mergeSort:
  it is unchanged in the recursive calls
Cannot use parameter xs:
  failed to eliminate recursive application
    mergeSort halves.fst


failed to prove termination, possible solutions:
  - Use `have`-expressions to prove the remaining goals
  - Use `termination_by` to specify a different well-founded relation
  - Use `decreasing_by` to specify your own tactic for discharging this kind of
goal
α : Type u_1
xs : List α
h : ¬xs.length < 2
halves : List α × List α := splitList xs
⊢ sizeOf (splitList xs).fst < sizeOf xs
```

The reason it terminates is that `splitList` always returns lists that are shorter than its input, at least when applied to lists that contain at least two elements. Thus, the length of `halves.fst` and `halves.snd` are less than the length of `xs`. This can be expressed using a `termination_by` clause:

```
def mergeSort [Ord α] (xs : List α) : List α :=
  if h : xs.length < 2 then
    match xs with
    | [] => []
    | [x] => [x]
  else
    let halves := splitList xs
    merge (mergeSort halves.fst) (mergeSort halves.snd)
termination_by xs.length
```

With this clause, the error message changes. Instead of complaining that the function isn't structurally recursive, Lean instead points out that it was unable to automatically prove that `(splitList xs).fst.length < xs.length`:

```
failed to prove termination, possible solutions:
  - Use `have`-expressions to prove the remaining goals
  - Use `termination_by` to specify a different well-founded relation
  - Use `decreasing_by` to specify your own tactic for discharging this kind of
goal
α : Type u_1
xs : List α
h : ¬xs.length < 2
halves : List α × List α := splitList xs
⊢ (splitList xs).fst.length < xs.length
```

Splitting a List Makes it Shorter

It will also be necessary to prove that `(splitList xs).snd.length < xs.length`. Because `splitList` alternates between adding entries to the two lists, it is easiest to prove both statements at once, so the structure of the proof can follow the algorithm used to implement `splitList`. In other words, it is easiest to prove that `∀(lst : List), (splitList lst).fst.length < lst.length ∧ (splitList lst).snd.length < lst.length`.

Unfortunately, the statement is false. In particular, `splitList []` is `([], [])`. Both output lists have length `0`, which is not less than `0`, the length of the input list. Similarly, `splitList ["basalt"]` evaluates to `(["basalt"], [])`, and `["basalt"]` is not shorter than `["basalt"]`. However, `splitList ["basalt", "granite"]` evaluates to `(["basalt"], ["granite"])`, and both of these output lists are shorter than the input list.

It turns out that the lengths of the output lists are always less than or equal to the length of the input list, but they are only strictly shorter when the input list contains at least two entries. It turns out to be easiest to prove the former statement, then extend it to the latter statement. Begin with a theorem statement:

```
theorem splitList_shorter_le (lst : List α) :
    (splitList lst).fst.length ≤ lst.length ∧
      (splitList lst).snd.length ≤ lst.length := by
  skip
```

```
unsolved goals
α : Type u_1
lst : List α
⊢ (splitList lst).fst.length ≤ lst.length ∧ (splitList lst).snd.length ≤
lst.length
```

Because `splitList` is structurally recursive on the list, the proof should use induction. The structural recursion in `splitList` fits a proof by induction perfectly: the base case of the induction matches the base case of the recursion, and the inductive step matches the recursive call. The `induction` tactic gives two goals:

```
theorem splitList_shorter_le (lst : List α) :
    (splitList lst).fst.length ≤ lst.length ∧
      (splitList lst).snd.length ≤ lst.length := by
  induction lst with
  | nil => skip
  | cons x xs ih => skip
```

```
unsolved goals
case nil
α : Type u_1
⊢ (splitList []).fst.length ≤ [].length ∧ (splitList []).snd.length ≤ [].length
```

```
unsolved goals
case cons
α : Type u_1
x : α
xs : List α
ih : (splitList xs).fst.length ≤ xs.length ∧ (splitList xs).snd.length ≤
xs.length
⊢ (splitList (x :: xs)).fst.length ≤ (x :: xs).length ∧ (splitList (x ::
xs)).snd.length ≤ (x :: xs).length
```

The goal for the `nil` case can be proved by invoking the simplifier and instructing it to unfold the definition of `splitList`, because the length of the empty list is less than or equal to the length of the empty list. Similarly, simplifying with `splitList` in the `cons` case places

`Nat.succ` around the lengths in the goal:

```
theorem splitList_shorter_le (lst : List α) :
    (splitList lst).fst.length ≤ lst.length ∧
      (splitList lst).snd.length ≤ lst.length := by
  induction lst with
  | nil => simp [splitList]
  | cons x xs ih =>
    simp [splitList]
```

```
unsolved goals
case cons
α : Type u_1
x : α
xs : List α
ih : (splitList xs).fst.length ≤ xs.length ∧ (splitList xs).snd.length ≤
xs.length
⊢ (splitList xs).snd.length ≤ xs.length ∧ (splitList xs).fst.length ≤ xs.length
+ 1
```

This is because the call to `List.length` consumes the head of the list `x :: xs`, converting it to a `Nat.succ`, in both the length of the input list and the length of the first output list.

Writing `A ∧ B` in Lean is short for `And A B`. `And` is a structure type in the `Prop` universe:

```
structure And (a b : Prop) : Prop where
  intro ::
  left : a
  right : b
```

In other words, a proof of `A ∧ B` consists of the `And.intro` constructor applied to a proof of `A` in the `left` field and a proof of `B` in the `right` field.

The `cases` tactic allows a proof to consider each constructor of a datatype or each potential proof of a proposition in turn. It corresponds to a `match` expression without recursion. Using `cases` on a structure results in the structure being broken apart, with an assumption added for each field of the structure, just as a pattern match expression extracts the field of a structure for use in a program. Because structures have only one constructor, using `cases` on a structure does not result in additional goals.

Because `ih` is a proof of `List.length (splitList xs).fst ≤ List.length xs ∧ List.length (splitList xs).snd ≤ List.length xs`, using `cases ih` results in an assumption that `List.length (splitList xs).fst ≤ List.length xs` and an assumption that `List.length (splitList xs).snd ≤ List.length xs`:

```
theorem splitList_shorter_le (lst : List α) :
    (splitList lst).fst.length ≤ lst.length ∧
      (splitList lst).snd.length ≤ lst.length := by
  induction lst with
  | nil => simp [splitList]
  | cons x xs ih =>
    simp [splitList]
    cases ih
```

```
unsolved goals
case cons.intro
α : Type u_1
x : α
xs : List α
left✝ : (splitList xs).fst.length ≤ xs.length
right✝ : (splitList xs).snd.length ≤ xs.length
⊢ (splitList xs).snd.length ≤ xs.length ∧ (splitList xs).fst.length ≤ xs.length
+ 1
```

Because the goal of the proof is also an `And`, the `constructor` tactic can be used to apply `And.intro`, resulting in a goal for each argument:

```
theorem splitList_shorter_le (lst : List α) :
    (splitList lst).fst.length ≤ lst.length ∧
      (splitList lst).snd.length ≤ lst.length := by
  induction lst with
  | nil => simp [splitList]
  | cons x xs ih =>
    simp [splitList]
    cases ih
    constructor
```

```
unsolved goals
case cons.intro.left
α : Type u_1
x : α
xs : List α
left✝ : (splitList xs).fst.length ≤ xs.length
right✝ : (splitList xs).snd.length ≤ xs.length
⊢ (splitList xs).snd.length ≤ xs.length

case cons.intro.right
α : Type u_1
x : α
xs : List α
left✝ : (splitList xs).fst.length ≤ xs.length
right✝ : (splitList xs).snd.length ≤ xs.length
⊢ (splitList xs).fst.length ≤ xs.length + 1
```

The `left` goal is identical to the `left✝` assumption, so the `assumption` tactic dispatches it:

```
theorem splitList_shorter_le (lst : List α) :
    (splitList lst).fst.length ≤ lst.length ∧
      (splitList lst).snd.length ≤ lst.length := by
  induction lst with
  | nil => simp [splitList]
  | cons x xs ih =>
    simp [splitList]
    cases ih
    constructor
    case left => assumption
```

```
unsolved goals
case cons.intro.right
α : Type u_1
x : α
xs : List α
left✝ : (splitList xs).fst.length ≤ xs.length
right✝ : (splitList xs).snd.length ≤ xs.length
⊢ (splitList xs).fst.length ≤ xs.length + 1
```

The `right` goal resembles the `right✝` assumption, except the goal adds a `+ 1` only to the length of the input list. It's time to prove that the inequality holds.

Adding One to the Greater Side

The inequality needed to prove `splitList_shorter_le` is `∀(n m : Nat), n ≤ m → n ≤ Nat.succ m`. The incoming assumption that `n ≤ m` essentially tracks the difference between `n` and `m` in the number of `Nat.le.step` constructors. Thus, the proof should add an extra `Nat.le.step` in the base case.

Starting out, the statement reads:

```
theorem Nat.le_succ_of_le : n ≤ m → n ≤ m + 1 := by
  skip
```

```
unsolved goals
n m : Nat
⊢ n ≤ m → n ≤ m + 1
```

The first step is to introduce a name for the assumption that `n ≤ m`:

```
theorem Nat.le_succ_of_le : n ≤ m → n ≤ m + 1 := by
  intro h
```

```
unsolved goals
n m : Nat
h : n ≤ m
⊢ n ≤ m + 1
```

The proof is by induction on this assumption:

```
theorem Nat.le_succ_of_le : n ≤ m → n ≤ m + 1 := by
  intro h
  induction h with
  | refl => skip
  | step _ ih => skip
```

In the case for `refl`, where `n = m`, the goal is to prove that `n ≤ n + 1`:

```
unsolved goals
case refl
n m : Nat
⊢ n ≤ n + 1
```

In the case for `step`, the goal is to prove that `n ≤ m + 1` under the assumption that `n ≤ m`:

```
unsolved goals
case step
n m m✝ : Nat
a✝ : n.le m✝
ih : n ≤ m✝ + 1
⊢ n ≤ m✝.succ + 1
```

For the `refl` case, the `step` constructor can be applied:

```
theorem Nat.le_succ_of_le : n ≤ m → n ≤ m + 1 := by
  intro h
  induction h with
  | refl => constructor
  | step _ ih => skip
```

```
unsolved goals
case refl.a
n m : Nat
⊢ n.le n
```

After `step`, `refl` can be used, which leaves only the goal for `step`:

```
theorem Nat.le_succ_of_le : n ≤ m → n ≤ m + 1 := by
  intro h
  induction h with
  | refl => constructor; constructor
  | step _ ih => skip
```

```
unsolved goals
case step
n m m✝ : Nat
a✝ : n.le m✝
ih : n ≤ m✝ + 1
⊢ n ≤ m✝.succ + 1
```

For the step, applying the `step` constructor transforms the goal into the induction hypothesis:

```
theorem Nat.le_succ_of_le : n ≤ m → n ≤ m + 1 := by
  intro h
  induction h with
  | refl => constructor; constructor
  | step _ ih => constructor
```

```
unsolved goals
case step.a
n m m✝ : Nat
a✝ : n.le m✝
ih : n ≤ m✝ + 1
⊢ n.le (m✝ + 1)
```

The final proof is as follows:

```
theorem Nat.le_succ_of_le : n ≤ m → n ≤ m + 1 := by
  intro h
  induction h with
  | refl => constructor; constructor
  | step => constructor; assumption
```

To reveal what's going on behind the scenes, the `apply` and `exact` tactics can be used to indicate exactly which constructor is being applied. The `apply` tactic solves the current goal by applying a function or constructor whose return type matches, creating new goals for each argument that was not provided, while `exact` fails if any new goals would be needed:

```
theorem Nat.le_succ_of_le : n ≤ m → n ≤ m + 1 := by
  intro h
  induction h with
  | refl => apply Nat.le.step; exact Nat.le.refl
  | step _ ih => apply Nat.le.step; exact ih
```

The proof can be golfed:

```
theorem Nat.le_succ_of_le (h : n ≤ m) : n ≤ m + 1:= by
  induction h <;> repeat (first | constructor | assumption)
```

In this short tactic script, both goals introduced by `induction` are addressed using `repeat (first | constructor | assumption)`. The tactic `first | T1 | T2 | ... | Tn` means to use try `T1` through `Tn` in order, using the first tactic that succeeds. In other words, `repeat (first | constructor | assumption)` applies constructors as long as it can, and then attempts to solve the goal using an assumption.

The proof can be shortened even further by using `omega`, a built-in solver for linear arithmetic:

```
theorem Nat.le_succ_of_le (h : n ≤ m) : n ≤ m + 1:= by
  omega
```

Finally, the proof can be written as a recursive function:

```
theorem Nat.le_succ_of_le : n ≤ m → n ≤ m + 1
  | .refl => .step .refl
  | .step h => .step (Nat.le_succ_of_le h)
```

Each style of proof can be appropriate to different circumstances. The detailed proof script is useful in cases where beginners may be reading the code, or where the steps of the proof provide some kind of insight. The short, highly-automated proof script is typically easier to maintain, because automation is frequently both flexible and robust in the face of small changes to definitions and datatypes. The recursive function is typically both harder to understand from the perspective of mathematical proofs and harder to maintain, but it can be a useful bridge for programmers who are beginning to work with interactive theorem proving.

Finishing the Proof

Now that both helper theorems have been proved, the rest of `splitList_shorter_le` will be completed quickly. The current proof state has one goal remaining:

```
unsolved goals
case cons.intro.right
α : Type u_1
x : α
xs : List α
left✝ : (splitList xs).fst.length ≤ xs.length
right✝ : (splitList xs).snd.length ≤ xs.length
⊢ (splitList xs).fst.length ≤ xs.length + 1
```

Using `Nat.le_succ_of_le` together with the `right✝` assumption completes the proof:

```
theorem splitList_shorter_le (lst : List α) :
    (splitList lst).fst.length ≤ lst.length ∧ (splitList lst).snd.length ≤
lst.length := by
  induction lst with
  | nil => simp [splitList]
  | cons x xs ih =>
    simp [splitList]
    cases ih
    constructor
    case left => assumption
    case right =>
      apply Nat.le_succ_of_le
      assumption
```

The next step is to return to the actual theorem that is needed to prove that merge sort terminates: that so long as a list has at least two entries, both results of splitting it are strictly shorter.

```
theorem splitList_shorter (lst : List α) (_ : lst.length ≥ 2) :
    (splitList lst).fst.length < lst.length ∧
      (splitList lst).snd.length < lst.length := by
  skip
```

```
unsolved goals
α : Type u_1
lst : List α
x✝ : lst.length ≥ 2
⊢ (splitList lst).fst.length < lst.length ∧ (splitList lst).snd.length <
lst.length
```

Pattern matching works just as well in tactic scripts as it does in programs. Because `lst` has at least two entries, they can be exposed with `match`, which also refines the type through dependent pattern matching:

```
theorem splitList_shorter (lst : List α) (_ : lst.length ≥ 2) :
    (splitList lst).fst.length < lst.length ∧
      (splitList lst).snd.length < lst.length := by
  match lst with
  | x :: y :: xs =>
    skip
```

```
unsolved goals
α : Type u_1
lst : List α
x y : α
xs : List α
x✝ : (x :: y :: xs).length ≥ 2
⊢ (splitList (x :: y :: xs)).fst.length < (x :: y :: xs).length ∧
    (splitList (x :: y :: xs)).snd.length < (x :: y :: xs).length
```

Simplifying using `splitList` removes `x` and `y`, resulting in the computed lengths of lists each gaining a `+ 1`:

```
theorem splitList_shorter (lst : List α) (_ : lst.length ≥ 2) :
    (splitList lst).fst.length < lst.length ∧
      (splitList lst).snd.length < lst.length := by
  match lst with
  | x :: y :: xs =>
    simp [splitList]
```

```
unsolved goals
α : Type u_1
lst : List α
x y : α
xs : List α
x✝ : (x :: y :: xs).length ≥ 2
⊢ (splitList xs).fst.length < xs.length + 1 ∧ (splitList xs).snd.length <
xs.length + 1
```

Replacing `simp` with `simp +arith` removes these `+ 1`s, because `simp +arith` makes use of the fact that `n + 1 < m + 1` implies `n < m`:

```
theorem splitList_shorter (lst : List α) (_ : lst.length ≥ 2) :
    (splitList lst).fst.length < lst.length ∧
      (splitList lst).snd.length < lst.length := by
  match lst with
  | x :: y :: xs =>
    simp +arith [splitList]
```

```
unsolved goals
α : Type u_1
lst : List α
x y : α
xs : List α
x✝ : (x :: y :: xs).length ≥ 2
⊢ (splitList xs).fst.length ≤ xs.length ∧ (splitList xs).snd.length ≤ xs.length
```

This goal now matches `splitList_shorter_le`, which can be used to conclude the proof:

```
theorem splitList_shorter (lst : List α) (_ : lst.length ≥ 2) :
    (splitList lst).fst.length < lst.length ∧
      (splitList lst).snd.length < lst.length := by
  match lst with
  | x :: y :: xs =>
    simp +arith [splitList]
    apply splitList_shorter_le
```

The facts needed to prove that `mergeSort` terminates can be pulled out of the resulting `And`:

```
theorem splitList_shorter_fst (lst : List α) (h : lst.length ≥ 2) :
    (splitList lst).fst.length < lst.length :=
  splitList_shorter lst h |>.left

theorem splitList_shorter_snd (lst : List α) (h : lst.length ≥ 2) :
    (splitList lst).snd.length < lst.length :=
  splitList_shorter lst h |>.right
```

Merge Sort Terminates

Merge sort has two recursive calls, one for each sub-list returned by `splitList`. Each recursive call will require a proof that the length of the list being passed to it is shorter than the length of the input list. It's usually convenient to write a termination proof in two steps: first, write down the propositions that will allow Lean to verify termination, and then prove them. Otherwise, it's possible to put a lot of effort into proving the propositions, only to find out that they aren't quite what's needed to establish that the recursive calls are on smaller inputs.

The `sorry` tactic can prove any goal, even false ones. It isn't intended for use in production code or final proofs, but it is a convenient way to "sketch out" a proof or program ahead of time. Any definitions or theorems that use `sorry` are annotated with a warning.

The initial sketch of `mergeSort`'s termination argument that uses `sorry` can be written by copying the goals that Lean couldn't prove into `have`-expressions. In Lean, `have` is similar to `let`. When using `have`, the name is optional. Typically, `let` is used to define names that refer to interesting values, while `have` is used to locally prove propositions that can be found when Lean is searching for evidence that an array lookup is in-bounds or that a function terminates.

```
def mergeSort [Ord α] (xs : List α) : List α :=
  if h : xs.length < 2 then
    match xs with
    | [] => []
    | [x] => [x]
  else
    let halves := splitList xs
    have : halves.fst.length < xs.length := by
      sorry
    have : halves.snd.length < xs.length := by
      sorry
    merge (mergeSort halves.fst) (mergeSort halves.snd)
termination_by xs.length
```

The warning is located on the name `mergeSort`:

```
declaration uses 'sorry'
```

Because there are no errors, the proposed propositions are enough to establish termination.

The proofs begin by applying the helper theorems:

```
def mergeSort [Ord α] (xs : List α) : List α :=
  if h : xs.length < 2 then
    match xs with
    | [] => []
    | [x] => [x]
  else
    let halves := splitList xs
    have : halves.fst.length < xs.length := by
      apply splitList_shorter_fst
    have : halves.snd.length < xs.length := by
      apply splitList_shorter_snd
    merge (mergeSort halves.fst) (mergeSort halves.snd)
termination_by xs.length
```

Both proofs fail, because `splitList_shorter_fst` and `splitList_shorter_snd` both require a proof that `xs.length ≥ 2`:

```
unsolved goals
case h
α : Type ?u.31067
inst✝ : Ord α
xs : List α
h : ¬xs.length < 2
halves : List α × List α := splitList xs
⊢ xs.length ≥ 2
```

To check that this will be enough to complete the proof, add it using `sorry` and check for errors:

```
def mergeSort [Ord α] (xs : List α) : List α :=
  if h : xs.length < 2 then
    match xs with
    | [] => []
    | [x] => [x]
  else
    let halves := splitList xs
    have : xs.length ≥ 2 := by sorry
    have : halves.fst.length < xs.length := by
      apply splitList_shorter_fst
      assumption
    have : halves.snd.length < xs.length := by
      apply splitList_shorter_snd
      assumption
    merge (mergeSort halves.fst) (mergeSort halves.snd)
termination_by xs.length
```

Once again, there is only a warning.

```
declaration uses 'sorry'
```

There is one promising assumption available: `h : ¬List.length xs < 2`, which comes from the `if`. Clearly, if it is not the case that `xs.length < 2`, then `xs.length ≥ 2`. The `omega` tactic solves this goal, and the program is now complete:

```
def mergeSort [Ord α] (xs : List α) : List α :=
  if h : xs.length < 2 then
    match xs with
    | [] => []
    | [x] => [x]
  else
    let halves := splitList xs
    have : xs.length ≥ 2 := by
      omega
    have : halves.fst.length < xs.length := by
      apply splitList_shorter_fst
      assumption
    have : halves.snd.length < xs.length := by
      apply splitList_shorter_snd
      assumption
    merge (mergeSort halves.fst) (mergeSort halves.snd)
termination_by xs.length
```

The function can be tested on examples:

```
#eval mergeSort ["soapstone", "geode", "mica", "limestone"]
```

```
["geode", "limestone", "mica", "soapstone"]
```

```
#eval mergeSort [5, 3, 22, 15]
```

```
[3, 5, 15, 22]
```

Division as Iterated Subtraction

Just as multiplication is iterated addition and exponentiation is iterated multiplication, division can be understood as iterated subtraction. The very first description of recursive functions in this book presents a version of division that terminates when the divisor is not zero, but that Lean does not accept. Proving that division terminates requires the use of a fact about inequalities.

Lean cannot prove that this definition of division terminates:

```
def div (n k : Nat) : Nat :=
  if n < k then
    0
  else
    1 + div (n - k) k
```

```
fail to show termination for
  div
with errors
failed to infer structural recursion:
Cannot use parameter n:
  failed to eliminate recursive application
    div (n - k) k
Cannot use parameter k:
  failed to eliminate recursive application
    div (n - k) k


Could not find a decreasing measure.
The basic measures relate at each recursive call as follows:
(<, ≤, =: relation proved, ? all proofs failed, _: no proof attempted)
           n k
1) 9:10-23 ≤ =
Please use `termination_by` to specify a decreasing measure.
```

That's a good thing, because it doesn't! When `k` is `0`, value of `n` does not decrease, so the program is an infinite loop.

Rewriting the function to take evidence that `k` is not `0` allows Lean to automaically prove termination:

```
def div (n k : Nat) (ok : k ≠ 0) : Nat :=
  if h : n < k then
    0
  else
    1 + div (n - k) k ok
```

This definition of `div` terminates because the first argument `n` is smaller on each recursive call. This can be expressed using a `termination_by` clause:

```
def div (n k : Nat) (ok : k ≠ 0) : Nat :=
  if h : n < k then
    0
  else
    1 + div (n - k) k ok
termination_by n
```

Exercises

Prove the following theorems:

- For all natural numbers n, $0 < n + 1$.
- For all natural numbers n, $0 \leq n$.
- For all natural numbers n and k, $(n + 1) - (k + 1) = n - k$
- For all natural numbers n and k, if $k < n$ then $n \neq 0$
- For all natural numbers n, $n - n = 0$
- For all natural numbers n and k, if $n + 1 < k$ then $n < k$

Safe Array Indices

The GetElem instance for Array and Nat requires a proof that the provided Nat is smaller than the array. In practice, these proofs often end up being passed to functions along with the indices. Rather than passing an index and a proof separately, a type called Fin can be used to bundle up the index and the proof into a single value. This can make code easier to read. Additionally, many of the built-in operations on arrays take their index arguments as Fin rather than as Nat, so using these built-in operations requires understanding how to use Fin.

The type Fin n represents numbers that are strictly less than n. In other words, Fin 3 describes 0, 1, and 2, while Fin 0 has no values at all. The definition of Fin resembles Subtype, as a Fin n is a structure that contains a Nat and a proof that it is less than n:

```
structure Fin (n : Nat) where
  val  : Nat
  isLt : LT.lt val n
```

Lean includes instances of ToString and OfNat that allow Fin values to be conveniently used as numbers. In other words, the output of #eval (5 : Fin 8) is 5, rather than something like {val := 5, isLt := _}.

Instead of failing when the provided number is larger than the bound, the OfNat instance for Fin returns a value modulo the bound. This means that #eval (45 : Fin 10) results in 5 rather than a compile-time error.

In a return type, a Fin returned as a found index makes its connection to the data structure in which it was found more clear. The Array.find in the previous section returns an index that the caller cannot immediately use to perform lookups into the array, because the information about its validity has been lost. A more specific type results in a value that can be used without making the program significantly more complicated:

```
def findHelper (arr : Array α) (p : α → Bool) (i : Nat) : Option (Fin arr.size ×
α) :=
  if h : i < arr.size then
    let x := arr[i]
    if p x then
      some (⟨i, h⟩, x)
    else findHelper arr p (i + 1)
  else none

def Array.find (arr : Array α) (p : α → Bool) : Option (Fin arr.size × α) :=
  findHelper arr p 0
```

Exercise

Write a function `Fin.next? : Fin n → Option (Fin n)` that returns the next largest `Fin` when it would be in bounds, or `none` if not. Check that

```
#eval (3 : Fin 8).next?
```

outputs

```
some 4
```

and that

```
#eval (7 : Fin 8).next?
```

outputs

```
none
```

Insertion Sort and Array Mutation

While insertion sort does not have the optimal worst-case time complexity for a sorting algorithm, it still has a number of useful properties:

- It is simple and straightforward to implement and understand
- It is an in-place algorithm, requiring no additional space to run
- It is a stable sort
- It is fast when the input is already almost sorted

In-place algorithms are particularly useful in Lean due to the way it manages memory. In some cases, operations that would normally copy an array can be optimized into mutation. This includes swapping elements in an array.

Most languages and run-time systems with automatic memory management, including JavaScript, the JVM, and .NET, use tracing garbage collection. When memory needs to be reclaimed, the system starts at a number of *roots* (such as the call stack and global values) and then determines which values can be reached by recursively chasing pointers. Any values that can't be reached are deallocated, freeing memory.

Reference counting is an alternative to tracing garbage collection that is used by a number of languages, including Python, Swift, and Lean. In a system with reference counting, each object in memory has a field that tracks how many references there are to it. When a new reference is established, the counter is incremented. When a reference ceases to exist, the counter is decremented. When the counter reaches zero, the object is immediately deallocated.

Reference counting has one major disadvantage compared to a tracing garbage collector: circular references can lead to memory leaks. If object A references object B , and object B references object A, they will never be deallocated, even if nothing else in the program references either A or B. Circular references result either from uncontrolled recursion or from mutable references. Because Lean supports neither, it is impossible to construct circular references.

Reference counting means that the Lean runtime system's primitives for allocating and deallocating data structures can check whether a reference count is about to fall to zero, and re-use an existing object instead of allocating a new one. This is particularly important when working with large arrays.

An implementation of insertion sort for Lean arrays should satisfy the following criteria:

1. Lean should accept the function without a `partial` annotation

2. If passed an array to which there are no other references, it should modify the array in-place rather than allocating a new one

The first criterion is easy to check: if Lean accepts the definition, then it is satisfied. The second, however, requires a means of testing it. Lean provides a built-in function called `dbgTraceIfShared` with the following signature:

```
#check dbgTraceIfShared
```

```
dbgTraceIfShared.{u} {α : Type u} (s : String) (a : α) : α
```

It takes a string and a value as arguments, and prints a message that uses the string to standard error if the value has more than one reference, returning the value. This is not, strictly speaking, a pure function. However, it is intended to be used only during development to check that a function is in fact able to re-use memory rather than allocating and copying.

When learning to use `dbgTraceIfShared`, it's important to know that `#eval` will report that many more values are shared than in compiled code. This can be confusing. It's important to build an executable with `lake` rather than experimenting in an editor.

Insertion sort consists of two loops. The outer loop moves a pointer from left to right across the array to be sorted. After each iteration, the region of the array to the left of the pointer is sorted, while the region to the right may not yet be sorted. The inner loop takes the element pointed to by the pointer and moves it to the left until the appropriate location has been found and the loop invariant has been restored. In other words, each iteration inserts the next element of the array into the appropriate location in the sorted region.

The Inner Loop

The inner loop of insertion sort can be implemented as a tail-recursive function that takes the array and the index of the element being inserted as arguments. The element being inserted is repeatedly swapped with the element to its left until either the element to the left is smaller or the beginning of the array is reached. The inner loop is structurally recursive on the `Nat` that is inside the `Fin` used to index into the array:

```
def insertSorted [Ord α] (arr : Array α) (i : Fin arr.size) : Array α :=
  match i with
  | ⟨0, _⟩ => arr
  | ⟨i' + 1, _⟩ =>
    have : i' < arr.size := by
      omega
    match Ord.compare arr[i'] arr[i] with
    | .lt | .eq => arr
    | .gt =>
      insertSorted (arr.swap i' i) ⟨i', by simp [*]⟩
```

If the index `i` is `0`, then the element being inserted into the sorted region has reached the beginning of the region and is the smallest. If the index is `i' + 1`, then the element at `i'` should be compared to the element at `i`. Note that while `i` is a `Fin arr.size`, `i'` is just a `Nat` because it results from the `val` field of `i`. Nonetheless, the proof automation used for checking array index notation includes `omega`, so `i'` is automatically usable as an index.

The two elements are looked up and compared. If the element to the left is less than or equal to the element being inserted, then the loop is finished and the invariant has been restored. If the element to the left is greater than the element being inserted, then the elements are swapped and the inner loop begins again. `Array.swap` takes both of its indices as `Nat`s, using the same tactics as array indexing behind the scenes to ensure that they are in bounds.

Nonetheless, the `Fin` used for the recursive call needs a proof that `i'` is in bounds for the result of swapping two elements. The `simp` tactic's database contains the fact that swapping two elements of an array doesn't change its size, and the `[*]` argument instructs it to additionally use the assumption introduced by `have`. Omitting the `have`-expression with the proof that `i' < arr.size` reveals the following goal:

```
unsolved goals
α : Type ?u.7
inst✝ : Ord α
arr : Array α
i : Fin arr.size
i' : Nat
isLt✝ : i' + 1 < arr.size
⊢ i' < arr.size
```

The Outer Loop

The outer loop of insertion sort moves the pointer from left to right, invoking `insertSorted` at each iteration to insert the element at the pointer into the correct position in the array.

The basic form of the loop resembles the implementation of `Array.map`:

```
def insertionSortLoop [Ord α] (arr : Array α) (i : Nat) : Array α :=
  if h : i < arr.size then
    insertionSortLoop (insertSorted arr ⟨i, h⟩) (i + 1)
  else
    arr
```

An error occurs because there is no argument that decreases at every recursive call:

```
fail to show termination for
  insertionSortLoop
with errors
failed to infer structural recursion:
Not considering parameter α of insertionSortLoop:
  it is unchanged in the recursive calls
Not considering parameter #2 of insertionSortLoop:
  it is unchanged in the recursive calls
Cannot use parameter arr:
  the type Array α does not have a `.brecOn` recursor
Cannot use parameter i:
  failed to eliminate recursive application
    insertionSortLoop (insertSorted arr ⟨i, h⟩) (i + 1)


Could not find a decreasing measure.
The basic measures relate at each recursive call as follows:
(<, ≤, =: relation proved, ? all proofs failed, _: no proof attempted)
            arr i #1
1) 473:6-57   ? ?  ?

#1: arr.size - i

Please use `termination_by` to specify a decreasing measure.
```

While Lean can prove that a `Nat` that increases towards a constant bound at each iteration leads to a terminating function, this function has no constant bound because the array is replaced with the result of calling `insertSorted` at each iteration.

Before constructing the termination proof, it can be convenient to test the definition with a `partial` modifier to make sure that it returns the expected answers:

```
partial def insertionSortLoop [Ord α] (arr : Array α) (i : Nat) : Array α :=
  if h : i < arr.size then
    insertionSortLoop (insertSorted arr ⟨i, h⟩) (i + 1)
  else
    arr
```

```
#eval insertionSortLoop #[5, 17, 3, 8] 0
```

```
#[3, 5, 8, 17]
```

```
#eval insertionSortLoop #["metamorphic", "igneous", "sedentary"] 0
```

```
#["igneous", "metamorphic", "sedentary"]
```

Termination

Once again, the function terminates because the difference between the index and the size of the array being processed decreases on each recursive call. This time, however, Lean does not accept the `termination_by`:

```
def insertionSortLoop [Ord α] (arr : Array α) (i : Nat) : Array α :=
  if h : i < arr.size then
    insertionSortLoop (insertSorted arr ⟨i, h⟩) (i + 1)
  else
    arr
termination_by arr.size - i
```

```
failed to prove termination, possible solutions:
  - Use `have`-expressions to prove the remaining goals
  - Use `termination_by` to specify a different well-founded relation
  - Use `decreasing_by` to specify your own tactic for discharging this kind of
goal
α : Type u_1
inst† : Ord α
arr : Array α
i : Nat
h : i < arr.size
⊢ (insertSorted arr ⟨i, h⟩).size - (i + 1) < arr.size - i
```

The problem is that Lean has no way to know that `insertSorted` returns an array that's the same size as the one it is passed. In order to prove that `insertionSortLoop` terminates, it is necessary to first prove that `insertSorted` doesn't change the size of the array. Copying the unproved termination condition from the error message to the function and "proving" it with `sorry` allows the function to be temporarily accepted:

```
def insertionSortLoop [Ord α] (arr : Array α) (i : Nat) : Array α :=
  if h : i < arr.size then
    have : (insertSorted arr ⟨i, h⟩).size - (i + 1) < arr.size - i := by
      sorry
    insertionSortLoop (insertSorted arr ⟨i, h⟩) (i + 1)
  else
    arr
termination_by arr.size - i
```

```
declaration uses 'sorry'
```

Because `insertSorted` is structurally recursive on the index of the element being inserted, the proof should be by induction on the index. In the base case, the array is returned unchanged, so its length certainly does not change. For the inductive step, the induction hypothesis is that a recursive call on the next smaller index will not change the length of the array. There are two cases two consider: either the element has been fully inserted into the sorted region and the array is returned unchanged, in which case the length is also unchanged, or the element is swapped with the next one before the recursive call. However, swapping two elements in an array doesn't change the size of it, and the induction hypothesis states that the recursive call with the next index returns an array that's the same size as its argument. Thus, the size remains unchanged.

Translating this English-language theorem statement to Lean and proceeding using the techniques from this chapter is enough to prove the base case and make progress in the inductive step:

```
theorem insert_sorted_size_eq [Ord α] (arr : Array α) (i : Fin arr.size) :
    (insertSorted arr i).size = arr.size := by
  match i with
  | ⟨j, isLt⟩ =>
    induction j with
    | zero => simp [insertSorted]
    | succ j' ih =>
      simp [insertSorted]
```

The simplification using `insertSorted` in the inductive step revealed the pattern match in `insertSorted`:

```
unsolved goals
case succc
α : Type u_1
inst✝ : Ord α
arr : Array α
i : Fin arr.size
j' : Nat
ih : ∀ (isLt : j' < arr.size), (insertSorted arr ⟨j', isLt⟩).size = arr.size
isLt : j' + 1 < arr.size
⊢ (match compare arr[j'] arr[j' + 1] with
      | Ordering.lt => arr
      | Ordering.eq => arr
      | Ordering.gt => insertSorted (arr.swap j' (j' + 1) ⋯ ⋯) ⟨j', ⋯⟩).size =
    arr.size
```

When faced with a goal that includes `if` or `match`, the `split` tactic (not to be confused with the `split` function used in the definition of merge sort) replaces the goal with one new goal for each path of control flow:

```
theorem insert_sorted_size_eq [Ord α] (arr : Array α) (i : Fin arr.size) :
    (insertSorted arr i).size = arr.size := by
  match i with
  | ⟨j, isLt⟩ =>
    induction j with
    | zero => simp [insertSorted]
    | succ j' ih =>
      simp [insertSorted]
      split
```

Because it typically doesn't matter *how* a statement was proved, but only *that* it was proved, proofs in Lean's output are typically replaced by `⋯`. Additionally, each new goal has an assumption that indicates which branch led to that goal, named `heq✝` in this case:

```
unsolved goals
case succ.h_1
α : Type u_1
inst✝ : Ord α
arr : Array α
i : Fin arr.size
j' : Nat
ih : ∀ (isLt : j' < arr.size), (insertSorted arr ⟨j', isLt⟩).size = arr.size
isLt : j' + 1 < arr.size
x✝ : Ordering
heq✝ : compare arr[j'] arr[j' + 1] = Ordering.lt
⊢ arr.size = arr.size

case succ.h_2
α : Type u_1
inst✝ : Ord α
arr : Array α
i : Fin arr.size
j' : Nat
ih : ∀ (isLt : j' < arr.size), (insertSorted arr ⟨j', isLt⟩).size = arr.size
isLt : j' + 1 < arr.size
x✝ : Ordering
heq✝ : compare arr[j'] arr[j' + 1] = Ordering.eq
⊢ arr.size = arr.size

case succ.h_3
α : Type u_1
inst✝ : Ord α
arr : Array α
i : Fin arr.size
j' : Nat
ih : ∀ (isLt : j' < arr.size), (insertSorted arr ⟨j', isLt⟩).size = arr.size
isLt : j' + 1 < arr.size
x✝ : Ordering
heq✝ : compare arr[j'] arr[j' + 1] = Ordering.gt
⊢ (insertSorted (arr.swap j' (j' + 1) ⋯ ⋯) ⟨j', ⋯⟩).size = arr.size
```

Rather than write proofs for both simple cases, adding `<;> try rfl` after `split` causes the two straightforward cases to disappear immediately, leaving only a single goal:

```
theorem insert_sorted_size_eq [Ord α] (arr : Array α) (i : Fin arr.size) :
    (insertSorted arr i).size = arr.size := by
  match i with
  | ⟨j, isLt⟩ =>
    induction j with
    | zero => simp [insertSorted]
    | succ j' ih =>
      simp [insertSorted]
      split <;> try rfl
```

```
unsolved goals
case succ.h_3
α : Type u_1
inst† : Ord α
arr : Array α
i : Fin arr.size
j' : Nat
ih : ∀ (isLt : j' < arr.size), (insertSorted arr ⟨j', isLt⟩).size = arr.size
isLt : j' + 1 < arr.size
x† : Ordering
heq† : compare arr[j'] arr[j' + 1] = Ordering.gt
⊢ (insertSorted (arr.swap j' (j' + 1) ⋯ ⋯) ⟨j', ⋯⟩).size = arr.size
```

Unfortunately, the induction hypothesis is not strong enough to prove this goal. The induction hypothesis states that calling `insertSorted` on `arr` leaves the size unchanged, but the proof goal is to show that the result of the recursive call with the result of swapping leaves the size unchanged. Successfully completing the proof requires an induction hypothesis that works for *any* array that is passed to `insertSorted` together with the smaller index as an argument

It is possible to get a strong induction hypothesis by using the `generalizing` option to the `induction` tactic. This option brings additional assumptions from the context into the statement that's used to generate the base case, the induction hypothesis, and the goal to be shown in the inductive step. Generalizing over `arr` leads to a stronger hypothesis:

```
theorem insert_sorted_size_eq [Ord α] (arr : Array α) (i : Fin arr.size) :
    (insertSorted arr i).size = arr.size := by
  match i with
  | ⟨j, isLt⟩ =>
    induction j generalizing arr with
    | zero => simp [insertSorted]
    | succ j' ih =>
      simp [insertSorted]
      split <;> try rfl
```

In the resulting goal, `arr` is now part of a "for all" statement in the inductive hypothesis:

```
unsolved goals
case succ.h_3
α : Type u_1
inst† : Ord α
j' : Nat
ih : ∀ (arr : Array α), Fin arr.size → ∀ (isLt : j' < arr.size), (insertSorted
arr ⟨j', isLt⟩).size = arr.size
arr : Array α
i : Fin arr.size
isLt : j' + 1 < arr.size
x† : Ordering
heq† : compare arr[j'] arr[j' + 1] = Ordering.gt
⊢ (insertSorted (arr.swap j' (j' + 1) ⋯ ⋯) ⟨j', ⋯⟩).size = arr.size
```

However, this whole proof is beginning to get unmanageable. The next step would be to introduce a variable standing for the length of the result of swapping, show that it is equal to `arr.size`, and then show that this variable is also equal to the length of the array that results from the recursive call. These equality statements can then be chained together to prove the goal. It's much easier, however, to carefully reformulate the theorem statement such that the induction hypothesis is automatically strong enough and the variables are already introduced. The reformulated statement reads:

```
theorem insert_sorted_size_eq [Ord α] (len : Nat) (i : Nat) :
    (arr : Array α) → (isLt : i < arr.size) → arr.size = len →
    (insertSorted arr ⟨i, isLt⟩).size = len := by
  skip
```

This version of the theorem statement is easier to prove for a few reasons:

1. Rather than bundling up the index and the proof of its validity in a `Fin`, the index comes before the array. This allows the induction hypothesis to naturally generalize over the array and the proof that `i` is in bounds.
2. An abstract length `len` is introduced to stand for `array.size`. Proof automation is often better at working with explicit statements of equality.

The resulting proof state shows the statement that will be used to generate the induction hypothesis, as well as the base case and the goal of the inductive step:

```
unsolved goals
α : Type u_1
inst† : Ord α
len i : Nat
⊢ ∀ (arr : Array α) (isLt : i < arr.size), arr.size = len → (insertSorted arr
⟨i, isLt⟩).size = len
```

Compare the statement with the goals that result from the `induction` tactic:

```
theorem insert_sorted_size_eq [Ord α] (len : Nat) (i : Nat) :
    (arr : Array α) → (isLt : i < arr.size) → arr.size = len →
    (insertSorted arr ⟨i, isLt⟩).size = len := by
  induction i with
  | zero => skip
  | succ i' ih => skip
```

In the base case, each occurrence of `i` has been replaced by `0`. Using `intro` to introduce each assumption and then simplifying using `insertSorted` will prove the goal, because `insertSorted` at index `zero` returns its argument unchanged:

```
unsolved goals
case zero
α : Type u_1
inst✝ : Ord α
len : Nat
⊢ ∀ (arr : Array α) (isLt : 0 < arr.size), arr.size = len → (insertSorted arr
⟨0, isLt⟩).size = len
```

In the inductive step, the induction hypothesis has exactly the right strength. It will be useful for *any* array, so long as that array has length `len`:

```
unsolved goals
case succ
α : Type u_1
inst✝ : Ord α
len i' : Nat
ih : ∀ (arr : Array α) (isLt : i' < arr.size), arr.size = len → (insertSorted
arr ⟨i', isLt⟩).size = len
⊢ ∀ (arr : Array α) (isLt : i' + 1 < arr.size), arr.size = len → (insertSorted
arr ⟨i' + 1, isLt⟩).size = len
```

In the base case, `simp` reduces the goal to `arr.size = len`:

```
theorem insert_sorted_size_eq [Ord α] (len : Nat) (i : Nat) :
    (arr : Array α) → (isLt : i < arr.size) → arr.size = len →
    (insertSorted arr ⟨i, isLt⟩).size = len := by
  induction i with
  | zero =>
    intro arr isLt hLen
    simp [insertSorted]
  | succ i' ih => skip
```

```
unsolved goals
case zero
α : Type u_1
inst✝ : Ord α
len : Nat
arr : Array α
isLt : 0 < arr.size
hLen : arr.size = len
⊢ arr.size = len
```

This can be proved using the assumption `hLen`. Adding the `*` parameter to `simp` instructs it to additionally use assumptions, which solves the goal:

```
theorem insert_sorted_size_eq [Ord α] (len : Nat) (i : Nat) :
    (arr : Array α) → (isLt : i < arr.size) → arr.size = len →
    (insertSorted arr ⟨i, isLt⟩).size = len := by
  induction i with
  | zero =>
    intro arr isLt hLen
    simp [insertSorted, *]
  | succ i' ih => skip
```

In the inductive step, introducing assumptions and simplifying the goal results once again in a goal that contains a pattern match:

```
theorem insert_sorted_size_eq [Ord α] (len : Nat) (i : Nat) :
    (arr : Array α) → (isLt : i < arr.size) → (arr.size = len) →
    (insertSorted arr ⟨i, isLt⟩).size = len := by
  induction i with
  | zero =>
    intro arr isLt hLen
    simp [insertSorted, *]
  | succ i' ih =>
    intro arr isLt hLen
    simp [insertSorted]
```

```
unsolved goals
case succ
α : Type u_1
inst✝ : Ord α
len i' : Nat
ih : ∀ (arr : Array α) (isLt : i' < arr.size), arr.size = len → (insertSorted
arr ⟨i', isLt⟩).size = len
arr : Array α
isLt : i' + 1 < arr.size
hLen : arr.size = len
⊢ (match compare arr[i'] arr[i' + 1] with
      | Ordering.lt => arr
      | Ordering.eq => arr
      | Ordering.gt => insertSorted (arr.swap i' (i' + 1) ⋯ ⋯) ⟨i', ⋯⟩).size =
    len
```

Using the `split` tactic results in one goal for each pattern. Once again, the first two goals result from branches without recursive calls, so the induction hypothesis is not necessary:

```
theorem insert_sorted_size_eq [Ord α] (len : Nat) (i : Nat) :
    (arr : Array α) → (isLt : i < arr.size) → (arr.size = len) →
    (insertSorted arr ⟨i, isLt⟩).size = len := by
  induction i with
  | zero =>
    intro arr isLt hLen
    simp [insertSorted, *]
  | succ i' ih =>
    intro arr isLt hLen
    simp [insertSorted]
    split
```

```
unsolved goals
case succ.h_1
α : Type u_1
inst✝ : Ord α
len i' : Nat
ih : ∀ (arr : Array α) (isLt : i' < arr.size), arr.size = len → (insertSorted
arr ⟨i', isLt⟩).size = len
arr : Array α
isLt : i' + 1 < arr.size
hLen : arr.size = len
x✝ : Ordering
heq✝ : compare arr[i'] arr[i' + 1] = Ordering.lt
⊢ arr.size = len

case succ.h_2
α : Type u_1
inst✝ : Ord α
len i' : Nat
ih : ∀ (arr : Array α) (isLt : i' < arr.size), arr.size = len → (insertSorted
arr ⟨i', isLt⟩).size = len
arr : Array α
isLt : i' + 1 < arr.size
hLen : arr.size = len
x✝ : Ordering
heq✝ : compare arr[i'] arr[i' + 1] = Ordering.eq
⊢ arr.size = len

case succ.h_3
α : Type u_1
inst✝ : Ord α
len i' : Nat
ih : ∀ (arr : Array α) (isLt : i' < arr.size), arr.size = len → (insertSorted
arr ⟨i', isLt⟩).size = len
arr : Array α
isLt : i' + 1 < arr.size
hLen : arr.size = len
x✝ : Ordering
heq✝ : compare arr[i'] arr[i' + 1] = Ordering.gt
⊢ (insertSorted (arr.swap i' (i' + 1) ⋯ ⋯) ⟨i', ⋯⟩).size = len
```

Running `try assumption` in each goal that results from `split` eliminates both of the non-recursive goals:

```
theorem insert_sorted_size_eq [Ord α] (len : Nat) (i : Nat) :
    (arr : Array α) → (isLt : i < arr.size) → (arr.size = len) →
    (insertSorted arr ⟨i, isLt⟩).size = len := by
  induction i with
  | zero =>
    intro arr isLt hLen
    simp [insertSorted, *]
  | succ i' ih =>
    intro arr isLt hLen
    simp [insertSorted]
    split <;> try assumption
```

```
unsolved goals
case succ.h_3
α : Type u_1
inst✝ : Ord α
len i' : Nat
ih : ∀ (arr : Array α) (isLt : i' < arr.size), arr.size = len → (insertSorted
arr ⟨i', isLt⟩).size = len
arr : Array α
isLt : i' + 1 < arr.size
hLen : arr.size = len
x✝ : Ordering
heq✝ : compare arr[i'] arr[i' + 1] = Ordering.gt
⊢ (insertSorted (arr.swap i' (i' + 1) ⋯ ⋯) ⟨i', ⋯⟩).size = len
```

The new formulation of the proof goal, in which a constant `len` is used for the lengths of all the arrays involved in the recursive function, falls nicely within the kinds of problems that `simp` can solve. This final proof goal can be solved by `simp [*]`, because the assumptions that relate the array's length to `len` are important:

```
theorem insert_sorted_size_eq [Ord α] (len : Nat) (i : Nat) :
    (arr : Array α) → (isLt : i < arr.size) → (arr.size = len) →
    (insertSorted arr ⟨i, isLt⟩).size = len := by
  induction i with
  | zero =>
    intro arr isLt hLen
    simp [insertSorted, *]
  | succ i' ih =>
    intro arr isLt hLen
    simp [insertSorted]
    split <;> try assumption
    simp [*]
```

Finally, because `simp [*]` can use assumptions, the `try assumption` line can be replaced by `simp [*]`, shortening the proof:

```
theorem insert_sorted_size_eq [Ord α] (len : Nat) (i : Nat) :
    (arr : Array α) → (isLt : i < arr.size) → (arr.size = len) →
    (insertSorted arr ⟨i, isLt⟩).size = len := by
  induction i with
  | zero =>
    intro arr isLt hLen
    simp [insertSorted, *]
  | succ i' ih =>
    intro arr isLt hLen
    simp [insertSorted]
    split <;> simp [*]
```

This proof can now be used to replace the `sorry` in `insertionSortLoop`. Providing `arr.size` as the `len` argument to the theorem causes the final conclusion to be `(insertSorted arr ⟨i, isLt⟩).size = arr.size`, so the rewrite ends with a very manageable proof goal:

```
def insertionSortLoop [Ord α] (arr : Array α) (i : Nat) : Array α :=
  if h : i < arr.size then
    have : (insertSorted arr ⟨i, h⟩).size - (i + 1) < arr.size - i := by
      rw [insert_sorted_size_eq arr.size i arr h rfl]
    insertionSortLoop (insertSorted arr ⟨i, h⟩) (i + 1)
  else
    arr
termination_by arr.size - i
```

```
unsolved goals
α : Type ?u.23737
inst✝ : Ord α
arr : Array α
i : Nat
h : i < arr.size
⊢ arr.size - (i + 1) < arr.size - i
```

The `omega` tactic can prove this:

```
def insertionSortLoop [Ord α] (arr : Array α) (i : Nat) : Array α :=
  if h : i < arr.size then
    have : (insertSorted arr ⟨i, h⟩).size - (i + 1) < arr.size - i := by
      rw [insert_sorted_size_eq arr.size i arr h rfl]
      omega
    insertionSortLoop (insertSorted arr ⟨i, h⟩) (i + 1)
  else
    arr
termination_by arr.size - i
```

The Driver Function

Insertion sort itself calls `insertionSortLoop`, initializing the index that demarcates the sorted region of the array from the unsorted region to `0`:

```
def insertionSort [Ord α] (arr : Array α) : Array α :=
  insertionSortLoop arr 0
```

A few quick tests show the function is at least not blatantly wrong:

```
#eval insertionSort #[3, 1, 7, 4]
```

```
#[1, 3, 4, 7]
```

```
#eval insertionSort #[ "quartz", "marble", "granite", "hematite"]
```

```
#["granite", "hematite", "marble", "quartz"]
```

Is This Really Insertion Sort?

Insertion sort is *defined* to be an in-place sorting algorithm. What makes it useful, despite its quadratic worst-case run time, is that it is a stable sorting algorithm that doesn't allocate extra space and that handles almost-sorted data efficiently. If each iteration of the inner loop allocated a new array, then the algorithm wouldn't *really* be insertion sort.

Lean's array operations, such as `Array.set` and `Array.swap`, check whether the array in question has a reference count that is greater than one. If so, then the array is visible to multiple parts of the code, which means that it must be copied. Otherwise, Lean would no longer be a pure functional language. However, when the reference count is exactly one, there are no other potential observers of the value. In these cases, the array primitives mutate the array in place. What other parts of the program don't know can't hurt them.

Lean's proof logic works at the level of pure functional programs, not the underlying implementation. This means that the best way to discover whether a program unnecessarily copies data is to test it. Adding calls to `dbgTraceIfShared` at each point where mutation is desired causes the provided message to be printed to `stderr` when the value in question has more than one reference.

Insertion sort has precisely one place that is at risk of copying rather than mutating: the call to `Array.swap`. Replacing `arr.swap ⟨i', by assumption⟩ i` with `((dbgTraceIfShared` `"array to swap" arr).swap ⟨i', by assumption⟩ i)` causes the program to emit `shared RC array to swap` whenever it is unable to mutate the array. However, this change to the program changes the proofs as well, because now there's a call to an additional function. Adding a local assumption that `dbgTraceIfShared` preserves the length of its argument and adding it to some calls to `simp` is enough to fix the program and proofs.

The complete instrumented code for insertion sort is:

```
def insertSorted [Ord α] (arr : Array α) (i : Fin arr.size) : Array α :=
  match i with
  | ⟨0, _⟩ => arr
  | ⟨i' + 1, _⟩ =>
    have : i' < arr.size := by
      omega
    match Ord.compare arr[i'] arr[i] with
    | .lt | .eq => arr
    | .gt =>
      have : (dbgTraceIfShared "array to swap" arr).size = arr.size := by
        simp [dbgTraceIfShared]
      insertSorted
        ((dbgTraceIfShared "array to swap" arr).swap i' i)
        ⟨i', by simp [*]⟩

theorem insert_sorted_size_eq [Ord α] (len : Nat) (i : Nat) :
    (arr : Array α) → (isLt : i < arr.size) → (arr.size = len) →
    (insertSorted arr ⟨i, isLt⟩).size = len := by
  induction i with
  | zero =>
    intro arr isLt hLen
    simp [insertSorted, *]
  | succ i' ih =>
    intro arr isLt hLen
    simp [insertSorted, dbgTraceIfShared]
    split <;> simp [*]

def insertionSortLoop [Ord α] (arr : Array α) (i : Nat) : Array α :=
  if h : i < arr.size then
    have : (insertSorted arr ⟨i, h⟩).size - (i + 1) < arr.size - i := by
      rw [insert_sorted_size_eq arr.size i arr h rfl]
      omega
    insertionSortLoop (insertSorted arr ⟨i, h⟩) (i + 1)
  else
    arr
termination_by arr.size - i

def insertionSort [Ord α] (arr : Array α) : Array α :=
  insertionSortLoop arr 0
```

A bit of cleverness is required to check whether the instrumentation actually works. First off, the Lean compiler aggressively optimizes function calls away when all their arguments are known at compile time. Simply writing a program that applies `insertionSort` to a large array is not sufficient, because the resulting compiled code may contain only the sorted array as a constant. The easiest way to ensure that the compiler doesn't optimize away the sorting routine is to read the array from `stdin`. Secondly, the compiler performs dead code elimination. Adding extra `let`s to the program won't necessarily result in more references in running code if the `let`-bound variables are never used. To ensure that the extra reference is not eliminated entirely, it's important to ensure that the extra reference is somehow used.

The first step in testing the instrumentation is to write `getLines`, which reads an array of lines from standard input:

```
def getLines : IO (Array String) := do
  let stdin ← IO.getStdin
  let mut lines : Array String := #[]
  let mut currLine ← stdin.getLine
  while !currLine.isEmpty do
     -- Drop trailing newline:
    lines := lines.push (currLine.dropRight 1)
    currLine ← stdin.getLine
  pure lines
```

`IO.FS.Stream.getLine` returns a complete line of text, including the trailing newline. It returns `""` when the end-of-file marker has been reached.

Next, two separate `main` routines are needed. Both read the array to be sorted from standard input, ensuring that the calls to `insertionSort` won't be replaced by their return values at compile time. Both then print to the console, ensuring that the calls to `insertionSort` won't be optimized away entirely. One of them prints only the sorted array, while the other prints both the sorted array and the original array. The second function should trigger a warning that `Array.swap` had to allocate a new array:

```
def mainUnique : IO Unit := do
  let lines ← getLines
  for line in insertionSort lines do
    IO.println line

def mainShared : IO Unit := do
  let lines ← getLines
  IO.println "--- Sorted lines: ---"
  for line in insertionSort lines do
    IO.println line

  IO.println ""
  IO.println "--- Original data: ---"
  for line in lines do
    IO.println line
```

The actual `main` simply selects one of the two main actions based on the provided command-line arguments:

```
def main (args : List String) : IO UInt32 := do
  match args with
  | ["--shared"] => mainShared; pure 0
  | ["--unique"] => mainUnique; pure 0
  | _ =>
    IO.println "Expected single argument, either \"--shared\" or \"--unique\""
    pure 1
```

Running it with no arguments produces the expected usage information:

```
$ sort
Expected single argument, either "--shared" or "--unique"
```

The file `test-data` contains the following rocks:

```
schist
feldspar
diorite
pumice
obsidian
shale
gneiss
marble
flint
```

Using the instrumented insertion sort on these rocks results them being printed in alphabetical order:

```
$ sort --unique < test-data
diorite
feldspar
flint
gneiss
marble
obsidian
pumice
schist
shale
```

However, the version in which a reference is retained to the original array results in a notification on `stderr` (namely, `shared RC array to swap`) from the first call to `Array.swap`:

```
$ sort --shared < test-data
shared RC array to swap
--- Sorted lines: ---
diorite
feldspar
flint
gneiss
marble
obsidian
pumice
schist
shale

--- Original data: ---
schist
feldspar
diorite
pumice
obsidian
shale
gneiss
marble
flint
```

The fact that only a single `shared RC` notification appears means that the array is copied only once. This is because the copy that results from the call to `Array.swap` is itself unique, so no further copies need to be made. In an imperative language, subtle bugs can result from forgetting to explicitly copy an array before passing it by reference. When running `sort --shared`, the array is copied as needed to preserve the pure functional meaning of Lean programs, but no more.

Other Opportunities for Mutation

The use of mutation instead of copying when references are unique is not limited to array update operators. Lean also attempts to "recycle" constructors whose reference counts are about to fall to zero, reusing them instead of allocating new data. This means, for instance, that `List.map` will mutate a linked list in place, at least in cases when nobody could possibly notice. One of the most important steps in optimizing hot loops in Lean code is making sure that the data being modified is not referred to from multiple locations.

Exercises

- Write a function that reverses arrays. Test that if the input array has a reference count of one, then your function does not allocate a new array.
- Implement either merge sort or quicksort for arrays. Prove that your implementation terminates, and test that it doesn't allocate more arrays than expected. This is a challenging exercise!

Special Types

Understanding the representation of data in memory is very important. Usually, the representation can be understood from the definition of a datatype. Each constructor corresponds to an object in memory that has a header that includes a tag and a reference count. The constructor's arguments are each represented by a pointer to some other object. In other words, `List` really is a linked list and extracting a field from a `structure` really does just chase a pointer.

There are, however, some important exceptions to this rule. A number of types are treated specially by the compiler. For example, the type `UInt32` is defined as `Fin (2 ^ 32)`, but it is replaced at run-time with an actual native implementation based on machine words. Similarly, even though the definition of `Nat` suggests an implementation similar to `List Unit`, the actual run-time representation uses immediate machine words for sufficiently-small numbers and an efficient arbitrary-precision arithmetic library for larger numbers. The Lean compiler translates from definitions that use pattern matching into the appropriate operations for this representation, and calls to operations like addition and subtraction are mapped to fast operations from the underlying arithmetic library. After all, addition should not take time linear in the size of the addends.

The fact that some types have special representations also means that care is needed when working with them. Most of these types consist of a `structure` that is treated specially by the compiler. With these structures, using the constructor or the field accessors directly can trigger an expensive conversion from an efficient representation to a slow one that is convenient for proofs. For example, `String` is defined as a structure that contains a list of characters, but the run-time representation of strings uses UTF-8, not linked lists of pointers to characters. Applying the constructor to a list of characters creates a byte array that encodes them in UTF-8, and accessing the field of the structure takes time linear in the length of the string to decode the UTF-8 representation and allocate a linked list. Arrays are represented similarly. From the logical perspective, arrays are structures that contain a list of array elements, but the run-time representation is a dynamically-sized array. At run time, the constructor translates the list into an array, and the field accessor allocates a linked list from the array. The various array operations are replaced with efficient versions by the compiler that mutate the array when possible instead of allocating a new one.

Both types themselves and proofs of propositions are completely erased from compiled code. In other words, they take up no space, and any computations that might have been performed as part of a proof are similarly erased. This means that proofs can take advantage of the convenient interface to strings and arrays as inductively-defined lists, including using induction to prove things about them, without imposing slow conversion steps while the program is running. For these built-in types, a convenient logical representation of the data does not imply that the program must be slow.

If a structure type has only a single non-type non-proof field, then the constructor itself disappears at run time, being replaced with its single argument. In other words, a subtype is represented identically to its underlying type, rather than with an extra layer of indirection. Similarly, `Fin` is just `Nat` in memory, and single-field structures can be created to keep track of different uses of `Nat` s or `String` s without paying a performance penalty. If a constructor has no non-type non-proof arguments, then the constructor also disappears and is replaced with a constant value where the pointer would otherwise be used. This means that `true` , `false` , and `none` are constant values, rather than pointers to heap-allocated objects.

The following types have special representations:

Type	Logical representation	Run-time Representation
`Nat`	Unary, with one pointer from each `Nat.succ`	Efficient arbitrary-precision integers
`Int`	A sum type with constructors for positive or negative values, each containing a `Nat`	Efficient arbitrary-precision integers
`BitVec w`	A `Fin` with an appropriate bound 2^w	Efficient arbitrary-precision integers
`UInt8` , `UInt16` , `UInt32` , `UInt64` , `USize`	A bitvector of the correct width	Fixed-precision machine integers
`Int8` , `Int16` , `Int32` , `Int64` , `ISize`	A wrapped unsigned integer of the same width	Fixed-precision machine integers
`Char`	A `UInt32` paired with a proof that it's a valid code point	Ordinary characters
`String`	A structure that contains a `List Char` in a field called `data`	UTF-8-encoded string
`Array α`	A structure that contains a `List α` in a field called `data`	Packed arrays of pointers to `α` values
`Sort u`	A type	Erased completely
Proofs of propositions	Whatever data is suggested by the proposition when considered as a type of evidence	Erased completely

Exercise

The definition of `Pos` does not take advantage of Lean's compilation of `Nat` to an efficient type. At run time, it is essentially a linked list. Alternatively, a subtype can be defined that allows Lean's fast `Nat` type to be used internally, as described in the initial section on subtypes. At run time, the proof will be erased. Because the resulting structure has only a single data field, it is represented as that field, which means that this new representation of `Pos` is identical to that of `Nat`.

After proving the theorem `∀ {n k : Nat}, n ≠ 0 → k ≠ 0 → n + k ≠ 0`, define instances of `ToString`, and `Add` for this new representation of `Pos`. Then, define an instance of `Mul`, proving any necessary theorems along the way.

Summary

Tail Recursion

Tail recursion is recursion in which the results of recursive calls are returned immediately, rather than being used in some other way. These recursive calls are called *tail calls*. Tail calls are interesting because they can be compiled to a jump instruction rather than a call instruction, and the current stack frame can be re-used instead of pushing a new frame. In other words, tail-recursive functions are actually loops.

A common way to make a recursive function faster is to rewrite it in accumulator-passing style. Instead of using the call stack to remember what is to be done with the result of a recursive call, an additional argument called an *accumulator* is used to collect this information. For example, an accumulator for a tail-recursive function that reverses a list contains the already-seen list entries, in reverse order.

In Lean, only self-tail-calls are optimized into loops. In other words, two functions that each end with a tail call to the other will not be optimized.

Reference Counting and In-Place Updates

Rather than using a tracing garbage collector, as is done in Java, C#, and most JavaScript implementations, Lean uses reference counting for memory management. This means that each value in memory contains a field that tracks how many other values refer to it, and the run-time system maintains these counts as references appear or disappear. Reference counting is also used in Python, PHP, and Swift.

When asked to allocate a fresh object, Lean's run-time system is able to recycle existing objects whose reference counts are falling to zero. Additionally, array operations such as `Array.set` and `Array.swap` will mutate an array if its reference count is one, rather than allocating a modified copy. If `Array.swap` holds the only reference to an array, then no other part of the program can tell that it was mutated rather than copied.

Writing efficient code in Lean requires the use of tail recursion and being careful to ensure that large arrays are used uniquely. While tail calls can be identified by inspecting the function's definition, understanding whether a value is referred to uniquely may require reading the whole program. The debugging helper `dbgTraceIfShared` can be used at key locations in the program to check that a value is not shared.

Proving Programs Correct

Rewriting a program in accumulator-passing style, or making other transformations that make it run faster, can also make it more difficult to understand. It can be useful to keep the original version of the program that is more clearly correct, and then use it as an executable specification for the optimized version. While techniques such as unit testing work just as well in Lean as in any other language, Lean also enables the use of mathematical proofs that completely ensure that both versions of the function return the same result for *all possible* inputs.

Typically, proving that two functions are equal is done using function extensionality (the `funext` tactic), which is the principle that two functions are equal if they return the same values for every input. If the functions are recursive, then induction is usually a good way to prove that their outputs are the same. Usually, the recursive definition of the function will make recursive calls on one particular argument; this argument is a good choice for induction. In some cases, the induction hypothesis is not strong enough. Fixing this problem usually requires thought about how to construct a more general version of the theorem statement that provides induction hypotheses that are strong enough. In particular, to prove that a function is equivalent to an accumulator-passing version, a theorem statement that relates arbitrary initial accumulator values to the final result of the original function is needed.

Safe Array Indices

The type `Fin n` represents natural numbers that are strictly less than `n`. `Fin` is short for "finite". As with subtypes, a `Fin n` is a structure that contains a `Nat` and a proof that this `Nat` is less than `n`. There are no values of type `Fin 0`.

If `arr` is an `Array α`, then `Fin arr.size` always contains a number that is a suitable index into `arr`. Many of the built-in array operators, such as `Array.swap`, take `Fin` values as arguments rather than separated proof objects.

Lean provides instances of most of the useful numeric type classes for `Fin`. The `OfNat` instances for `Fin` perform modular arithmetic rather than failing at compile time if the number provided is larger than the `Fin` can accept.

Provisional Proofs

Sometimes, it can be useful to pretend that a statement is proved without actually doing the work of proving it. This can be useful when making sure that a proof of a statement would be suitable for some task, such as a rewrite in another proof, determining that an array access is safe, or showing that a recursive call is made on a smaller value than the original argument. It's very frustrating to spend time proving something, only to discover that some other proof would have been more useful.

The `sorry` tactic causes Lean to provisionally accept a statement as if it were a real proof. It can be seen as analogous to a stub method that throws a `NotImplementedException` in C#. Any proof that relies on `sorry` includes a warning in Lean.

Be careful! The `sorry` tactic can prove *any* statement, even false statements. Proving that `3 < 2` can cause an out-of-bounds array access to persist to runtime, unexpectedly crashing a program. Using `sorry` is convenient during development, but keeping it in the code is dangerous.

Proving Termination

When a recursive function does not use structural recursion, Lean cannot automatically determine that it terminates. In these situations, the function could just be marked `partial`. However, it is also possible to provide a proof that the function terminates.

Partial functions have a key downside: they can't be unfolded during type checking or in proofs. This means that Lean's value as an interactive theorem prover can't be applied to them. Additionally, showing that a function that is expected to terminate actually always does terminate removes one more potential source of bugs.

The `termination_by` clause that's allowed at the end of a function can be used to specify the reason why a recursive function terminates. The clause maps the function's arguments to an expression that is expected to be smaller for each recursive call. Some examples of expressions that might decrease are the difference between a growing index into an array and the array's size, the length of a list that's cut in half at each recursive call, or a pair of lists, exactly one of which shrinks on each recursive call.

Lean contains proof automation that can automatically determine that some expressions shrink with each call, but many interesting programs will require manual proofs. These proofs can be provided with `have`, a version of `let` that's intended for locally providing proofs rather than values.

A good way to write recursive functions is to begin by declaring them `partial` and debugging them with testing until they return the right answers. Then, `partial` can be removed and replaced with a `termination_by` clause. Lean will place error highlights on each recursive call for which a proof is needed that contains the statement that needs to be proved. Each of these statements can be placed in a `have`, with the proof being `sorry`. If Lean accepts the program and it still passes its tests, the final step is to actually prove the theorems that enable Lean to accept it. This approach can prevent wasting time on proving that a buggy program terminates.

Next Steps

This book introduces the very basics of functional programming in Lean, including a tiny amount of interactive theorem proving. Using dependently-typed functional languages like Lean is a deep topic, and much can be said. Depending on your interests, the following resources might be useful for learning Lean 4.

Learning Lean

Lean 4 itself is described in the following resources:

- Theorem Proving in Lean 4 is a tutorial on writing proofs using Lean.
- The Lean 4 Manual provides a detailed description of the language and its features.
- How To Prove It With Lean is a Lean-based accompaniment to the well-regarded textbook *How To Prove It* that provides an introduction to writing paper-and-pencil mathematical proofs.
- Metaprogramming in Lean 4 provides an overview of Lean's extension mechanisms, from infix operators and notations to macros, custom tactics, and full-on custom embedded languages.
- Functional Programming in Lean may be interesting to readers who enjoy jokes about recursion.

However, the best way to continue learning Lean is to start reading and writing code, consulting the documentation when you get stuck. Additionally, the Lean Zulip is an excellent place to meet other Lean users, ask for help, and help others.

Mathematics in Lean

A wide selection of learning resources for mathematicians are available at the community site.

Using Dependent Types in Computer Science

Rocq is a language that has a lot in common with Lean. For computer scientists, the Software Foundations series of interactive textbooks provides an excellent introduction to applications of Rocq in computer science. The fundamental ideas of Lean and Rocq are very similar, and skills are readily transferable between the systems.

Programming with Dependent Types

For programmers who are interested in learning to use indexed families and dependent types to structure programs, Edwin Brady's *Type Driven Development with Idris* provides an excellent introduction. Like Rocq, Idris is a close cousin of Lean, though it lacks tactics.

Understanding Dependent Types

The Little Typer is a book for programmers who haven't formally studied logic or the theory of programming languages, but who want to build an understanding of the core ideas of dependent type theory. While all of the above resources aim to be as practical as possible, *The Little Typer* presents an approach to dependent type theory where the very basics are built up from scratch, using only concepts from programming. Disclaimer: the author of *Functional Programming in Lean* is also an author of *The Little Typer*.