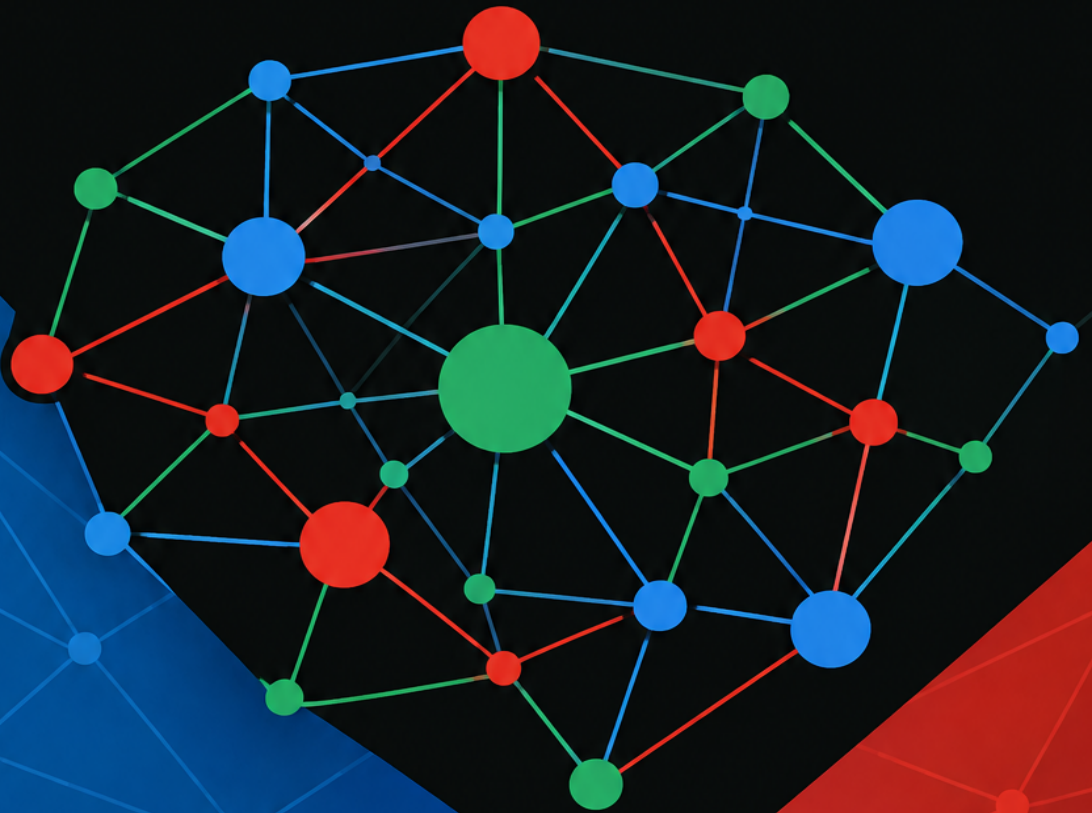


Lean 4:

Programming, Proof, and
Formal Verification



Herbert D. Collins

Lean 4: Programming, Proof, and Formal Verification

A Deep Guide to Functional Programming and Interactive Theorem Proving

Steve Publications

This book is available at

<https://leanpub.com/lean4programmingproofandformalverification>

This version was published on 2026-10-02



This is a [Leanpub](#) book. Leanpub empowers authors and publishers with the Lean Publishing process. [Lean Publishing](#) is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

© 2026 Steve Publications

Contents

A Deep Guide to Functional Programming and Interactive Theorem Proving	1
Chapter 1: Why Lean	2
The Problem of Correctness in Software and Mathematics	2
Unifying Programs and Proofs	3
A Brief History of Lean and Its Design Philosophy	4
What Lean Can Do: Use Cases and Examples	6
How to Read This Book	8
Chapter 2: Getting Started with Lean 4	11
Installing Lean and Lake	11
Your First Lean Project	12
Editors, LSP and the Lean Environment	14
Running and Interacting with Lean	15
The Lake Build System and Package Manager	18
Project Structure and Configuration	20
Chapter 3: The Lean Syntax and Basic Types	24
Modules, Namespaces and Imports	24
Identifiers and Declarations	24
Basic Types and Literals	24
Variables and Local Definitions	24
Functions and Applications	24
Pattern Matching	24
Chapter 4: Dependent Types and the Foundations	26
From Simple Types to Dependent Types	26
The Calculus of Constructions	26
Universes and Polymorphism	26
Pi Types and Dependent Functions	26

CONTENTS

Dependent Pairs and Sigma Types	26
Definitional Equality and Conversion	26
Chapter 5: Inductive Types and Data Structures	28
Defining Inductive Types	28
Pattern Matching and Recursion	28
Lists, Vectors and Indexed Types	28
Inductive Families	28
Algebraic Data Types in Practice	28
Recursive and Well-founded Functions	28
Chapter 6: Functional Programming in Lean	30
Higher-order Functions and Composition	30
Type Classes and Generic Programming	30
Structures and Records	30
Monads and Do Notation	30
Standard Library Collections	30
Performance and Mutation	30
Chapter 7: Propositions as Types	32
Prop and Type: Two Universes of Discourse	32
Logical Connectives as Types	32
Quantifiers as Dependent Types	32
Proof Terms and Proof Checking	32
Classical and Constructive Logic	32
Decidability and Computation	32
Chapter 8: Writing Proofs in Lean	34
Term-Style Proofs	34
Tactic Mode Basics	34
Core Tactics: intro, apply, exact, refine	34
Breaking Down Goals: constructor, cases, induction	34
Structuring Proofs: have, show, calc	34
When to Use Tactics Versus Terms	34
Chapter 9: Equality and Rewriting	36
Definitional and Propositional Equality	36
The Equality Type and Its Eliminator	36
The rw and subst Tactics	36
Calc Blocks for Chain Reasoning	36

CONTENTS

Function Extensionality and Congruence	36
Quotients and Equivalence Relations	36
Chapter 10: Automation and Simplification	38
The Simplifier: simp and simp_lemmas	38
Rewrite Tactics and Strategies	38
Decision Procedures: decide and native_decide	38
Arithmetic Automation	38
Proof Search: aesop and Other Tools	38
When Automation Helps and Hurts	38
Chapter 11: The Lean Type System Deep Dive	40
Lean's Type Theory: A Formal View	40
Inductive Type Recursors and Eliminators	40
Computation Rules and Reduction	40
Termination and Well-founded Recursion	40
Universe Levels and Size Constraints	40
Type Checking and Soundness	40
Chapter 12: Implicit Arguments and Type Class Resolution	42
Implicit Arguments and Inference	42
Type Classes: Concept and Mechanism	42
Instance Resolution and Search	42
Coercions and Canonical Constructions	42
Type Class Hierarchies and Algebra	42
Debugging Type Class Failures	42
Chapter 13: The Lean Architecture	44
The Trusted Computing Base	44
From Source to Proof Term: The Elaboration Pipeline	44
The Parser and Syntax Trees	44
The Kernel: Proof Checking	44
The Compiler and Runtime	44
Untrusted but Useful: Automation and Tooling	44
Chapter 14: Metaprogramming and Extensibility	46
Why Extend Lean?	46
Syntax Extensions and Custom Notation	46
Hygienic Macros	46
Elaborators and Commands	46

Tactic Implementation	46
Building Domain-Specific Languages	46
Chapter 15: Formal Verification and Verified Programming	48
Specification: Pre-, Post- and Invariants	48
Verifying Algorithms and Data Structures	48
Functional Correctness and Properties	48
Verified Compilers and Translations	48
Cryptographic and Security Properties	48
Case Study: A Verified Data Structure	48
Chapter 16: Mathematical Formalization with Mathlib	50
Lean Core, Stdlib and Mathlib	50
Working with Existing Formalizations	50
Algebra and Type Class Hierarchies	50
Analysis and Topology	50
Discrete Mathematics and Combinatorics	50
Engineering Large Formalizations	50
Chapter 17: Practical Engineering and Best Practices	52
Project Organization at Scale	52
Namespace and Module Design	52
Performance Engineering	52
Documentation and Readability	52
Testing and Debugging Proofs	52
Common Pitfalls and Solutions	52
Chapter 18: The Lean Ecosystem and Beyond	54
Lean 4 Version History and Releases	54
Tooling and Integrations	54
Community and Contribution	54
Interoperability and FFIs	54
Real-World Deployments	54
Where Lean Is Headed	54
Conclusion	56
References	57

A Deep Guide to Functional Programming and Interactive Theorem Proving

This book provides a thorough, technically precise exploration of Lean 4 as a functional programming language and interactive theorem prover. It is intended for software engineers, mathematicians and researchers who want to understand Lean at a deep level: its type-theoretic foundations, its architecture, its facilities for programming and verification and its ecosystem for building large formal projects. By the end of this book you will be able to write executable Lean programs, develop machine-checked proofs, construct custom automation and language extensions and organize substantial formal verification projects. The material assumes familiarity with programming and basic mathematics but no prior background in type theory, dependent types, or theorem provers.

Chapter 1: Why Lean

The Problem of Correctness in Software and Mathematics

Consider a compiler that optimizes programs. It takes source code as input and produces machine code as output. Its job is to preserve the program's behavior while making it faster. A compiler bug can introduce subtle, devastating errors that evade testing and emerge in production systems. Now consider a formal proof in mathematics. It is a chain of logical deductions from premises to conclusion, verified by peer review and human scrutiny. A flaw in reasoning, however small, invalidates the entire result.

Both of these scenarios confront the same fundamental problem: how do we establish and maintain confidence that a complex, intricate process produces correct results? The answer in both cases is verification: systematically checking that the output matches the intended specification.

Traditional software testing covers only what it tests. The absence of failing tests says nothing about correctness on untested inputs. Static analysis tools detect classes of bugs but produce false positives and cannot verify functional correctness. Formal verification offers a different approach: mathematical proof that a program satisfies its specification for all possible inputs. When done right, the result is certainty rather than confidence.

Mathematics faces its own version of the problem. Mathematical proofs in natural language are necessarily imprecise. They rely on informal conventions, omitted details and implicit assumptions. Over decades, mathematicians have found errors in published proofs of results that stood for years: flawed proofs of the Kepler conjecture, gaps in proofs of major theorems in topology and algebra, incorrect statements in influential papers. Human verification is fallible and the complexity of modern mathematics exceeds what any single person can fully check.

Formal verification addresses both problems through a common mechanism: machine-checked proofs. The idea is simple in principle. Encode the

specification precisely in a formal language. Encode the system or proof as a derivation in that language. Use a computer program to check every step mechanically. The result is a proof whose correctness depends only on the correctness of the checking program, which can be made small and thoroughly examined.

This approach is not new. Automated theorem provers have existed for decades. But Lean represents a specific answer to a set of practical questions: how do we make formal verification scalable, how do we integrate it into real-world software development and how do we make it accessible to both engineers and mathematicians?

Unifying Programs and Proofs

Lean is built around a single fundamental insight: programs and proofs are the same thing. This statement needs explanation.

Lean is based on dependent type theory, a formal system in which types can depend on values. In such a system, a type can express not just what kind of data we are dealing with but what properties that data satisfies. For instance, we can write a type that denotes “arrays of length n ” rather than just “arrays”. We can write a type that denotes “the result of sorting this list” rather than just “a list”. We can write a type that denotes “a proof that this function terminates”.

In this framework, a function type is also an implication: a function of type $A \rightarrow B$ takes a value of type A and produces a value of type B . If we interpret types as propositions, then a function is a proof that proposition A implies proposition B . A value of type A is a proof of proposition A . The system of types and values encodes the entire logical apparatus of mathematics. This correspondence between logic and type theory is known as the Curry-Howard isomorphism or the propositions-as-types principle.

Lean exploits this correspondence directly. When you write a function in Lean, you are writing a program. When you write a proof in Lean, you are writing a term of the type that represents the proposition being proved. The same language constructs, the same type checker and the same reduction rules apply to both. A proof is data: it can be passed around, transformed, inspected and compiled.

This unity has practical consequences. First, Lean is a general-purpose functional programming language capable of expressing any computable func-

tion. The same language that you use to write algorithms and data structures is the language you use to write proofs. Second, because proofs are terms, they are subject to the same rigor as code: they type-check or they do not. Third, because programs can be annotated with types that express their correctness properties, the type checker verifies both that the program computes something and that it computes the right thing.

To see how this works concretely, consider a function that computes the length of a list and a proposition stating that appending two lists increases the length by the sum of their lengths:

```

1  def length : List Nat -> Nat
2    | [] => 0
3    | _ :: xs => 1 + length xs
4
5  theorem length_append : forall (l1 l2 : List Nat),
6    length (l1 ++ l2) = length l1 + length l2 := by
7    intro l1 l2
8    induction l1 with
9    | nil =>
10     simp [length]
11    | cons x xs ih =>
12     simp [length, ih, Nat.add_assoc]

```

The function `length` is executable code. You can run it on concrete lists and get results. The theorem `length_append` is a proof term of the type that states the property. The proof is constructed using tactics that manipulate the proof state, but what they produce is ultimately a term that type-checks against the theorem's statement. Lean's kernel independently verifies that this term is correct: no cleverness in the tactics can fool the kernel into accepting an invalid proof.

This distinction is critical and worth stating explicitly: tactics and automation in Lean generate proof terms. The kernel, a small, trusted component of Lean, checks those proof terms directly. The kernel does not know about tactics, simplifiers, or decision procedures. It knows only the basic rules of the type theory. This architecture means you can build powerful automation on top of Lean without expanding the trusted computing base. The automation may be wrong, but if it produces a term the kernel accepts, the proof is correct.

A Brief History of Lean and Its Design Philosophy

Lean was started in 2013 by Leonardo de Moura at Microsoft Research. The name is an acronym: Logical Engine for Applications. From the beginning, Lean was designed to address specific shortcomings of existing proof assistants.

Coq, Isabelle/HOL and Agda are the major alternatives, each with different strengths and weaknesses. Coq has one of the largest ecosystems and the most mature formal library, but its type theory is restrictive in ways that complicate certain developments. Isabelle/HOL uses classical higher-order logic with a powerful automation ecosystem but separates the logic from the programming language in ways that make interactive development less seamless. Agda has an elegant dependent type system but limited automation and a smaller ecosystem.

Lean's design decisions reflect a specific set of trade-offs. First, Lean's type theory is based on the Calculus of Constructions with inductive types and quotients, similar to Coq but with some differences. It has an impredicative Prop universe for propositions, allowing propositions to quantify over all propositions, which simplifies many logical developments. It has a predicative hierarchy of Type universes for data. It supports quotient types, which allow constructing types by identifying equivalent elements.

Second, Lean was designed from the start with a minimal trusted kernel. The kernel implements only the core type-checking rules. Everything else: tactics, simplifiers, decision procedures, macro systems, type class resolution, even most of the parser and elaborator, is implemented in Lean itself and verified only by the fact that it produces proof terms the kernel accepts. This design makes the system auditable: if you want to know what Lean trusts, you read the kernel code, which is small enough to read.

Third, Lean was designed for extensibility. Its syntax is extensible via macros and custom parser rules. Its tactics can be composed and extended in the Lean language itself. Type classes provide a flexible mechanism for ad-hoc polymorphism and library organization. The metaprogramming framework allows building domain-specific languages, custom automation and specialized notations. These design choices make Lean well-suited for large-scale developments in diverse areas.

Lean 1 and Lean 2 were implemented in OCaml and were largely prototypes. Lean 3, released in January 2017, was a major rewrite in C++ that introduced

many of the features that make Lean practical for large projects. Lean 3 supported the growth of Mathlib, a community-developed library of formalized mathematics that grew to over one million lines.

Lean 4, released in September 2023, is a complete rewrite that addresses fundamental limitations of Lean 3. Most significantly, Lean 4 is self-hosted: approximately 90 percent of its implementation is written in Lean itself rather than C++. This change has several consequences. It means the language and the meta-language are the same, simplifying development and extensibility. It means the system can be built from source without a complex C++ toolchain. It means the Lean implementation can be verified using Lean itself, though this verification is an ongoing project.

Lean 4 also introduced a new standard library, a new build system called Lake, improved performance and numerous changes to the syntax and semantics. It is not backward-compatible with Lean 3, which was a necessary cost of the redesign. The current stable version is in the 4.34.0 range (4.34.1) as of late 2026, with the language reference documenting version 4.35.0-rc3.

The project is maintained by Lean FRO, a nonprofit organization founded in July 2023 by Leonardo de Moura and Sebastian Ullrich, funded by the Simons Foundation International, the Alfred P. Sloan Foundation and Richard Merkin. The source code is open-source under the Apache 2.0 license and hosted at <https://github.com/leanprover/lean4>.

What Lean Can Do: Use Cases and Examples

Lean is used in two broad categories of projects: formalizing mathematics and verifying software. These categories overlap: software verification uses the same logical machinery as mathematical formalization and mathematical formalization often requires engineering sophisticated abstractions that are relevant to software development.

Mathematical formalization is where Lean has achieved some of its most visible successes. The community-developed library Mathlib4 contains formalizations of large swaths of undergraduate and graduate mathematics: group theory, ring theory, field theory, linear algebra, topology, measure theory, real analysis, complex analysis, probability, category theory, number theory and more. The library has grown past 1.5 million lines of Lean code. Recent milestones include:

- A formal proof of Fermat’s Last Theorem, completed independently by teams using Lean, including efforts by Anthropic and OpenAI that demonstrated the feasibility of AI-assisted formalization at the highest levels of mathematical complexity.
- A formal proof of the falsity of the Erdős unit-distance problem, submitted with over 1.2 million lines of Lean code.
- Large-scale formalizations of algebraic geometry, homotopy type theory and advanced topics in analysis and probability.

These are not trivial achievements. Formalizing mathematics requires translating mathematical arguments into machine-checked proofs, filling in omitted details, choosing appropriate representations and managing complexity across large developments. Lean’s type theory, automation tools and library infrastructure make this feasible at a scale that was previously difficult.

Software verification is where Lean’s dual nature as programming language and proof assistant is most directly relevant. Several high-profile projects use Lean to verify critical systems:

Cedar is an authorization policy language developed by AWS. It provides fine-grained access control for AWS services and is used in production in AWS Verified Permissions and AWS Verified Access. AWS scientists Emina Torlak and Kesha Hietala built a formal model of Cedar in Lean and proved key security properties of its design, including correctness of the authorization algorithm and safety of policy evaluation. Their approach, called verification-guided development, builds a Lean model that is approximately ten times smaller than the Rust implementation and uses differential testing to ensure the model and implementation agree. The Lean proofs are mandatory for every release.

Aeneas is a verification framework that translates Rust programs into Lean for verification. Microsoft uses Aeneas to verify SymCrypt, the cryptographic library that protects Windows, Azure, Xbox and other Microsoft products. SymCrypt is being rewritten from C to Rust and verified against cryptographic specifications including ML-KEM, the NIST post-quantum key encapsulation mechanism. The first verified implementations of ML-KEM and SHA-3 in SymCrypt are running in Windows Insider builds.

ArkLib is a library of formally verified specifications for zero-knowledge proof systems, contributed by Derek Sorensen of the Ethereum Foundation. It provides machine-checked specifications for interactive oracle reductions, sum-check protocols and FRI-based proving systems. Production Rust provers

can be verified against these specifications using Rust-to-Lean extraction tools.

These projects share a common pattern. They build an executable Lean model of the system, prove key properties of the model and ensure the implementation matches the model through testing or extraction. This pattern leverages Lean's strength: expressing precise specifications and proving properties about them.

Beyond these high-profile projects, Lean is used for verifying compilers, cryptographic protocols, operating system components, distributed algorithms and domain-specific systems. Its growing ecosystem of verification tools and libraries makes it increasingly practical for a wide range of formal verification tasks.

How to Read This Book

This book is organized to take you from fundamentals to advanced topics in a systematic progression. The chapters build on each other, so reading in order is recommended. However, you can use this book as a reference for specific topics once you have grasped the fundamentals.

The book is divided into four parts, though this division is implicit in the chapter ordering rather than explicit in the structure.

Chapters 1 through 3 establish the basics. Chapter 1 is this chapter: motivation, history and overview. Chapter 2 covers installation, tooling and your first Lean project. Chapter 3 introduces Lean's syntax, basic types, functions, pattern matching and organizational constructs like modules and namespaces. If you are new to Lean, read these chapters carefully before moving on.

Chapters 4 through 12 cover Lean's type system, programming facilities and proof mechanisms in depth. Chapter 4 introduces dependent types and the core type-theoretic foundations. Chapter 5 covers inductive types and recursive data structures. Chapter 6 explores Lean as a functional programming language. Chapter 7 explains the propositions-as-types correspondence. Chapters 8 and 9 cover proof writing: tactics, term-mode proofs, equality and rewriting. Chapter 10 discusses automation and simplification. Chapter 11 gives a technical treatment of the type system. Chapter 12 covers type classes and implicit arguments in detail. These chapters form the core of the book.

Chapters 13 through 16 focus on Lean’s architecture and its applications. Chapter 13 explains the internal architecture: the elaboration pipeline, the kernel, the compiler and the distinction between trusted and untrusted components. Chapter 14 covers metaprogramming: macros, syntax extensions, elaborators and custom tactics. Chapter 15 discusses formal verification and verified programming. Chapter 16 covers mathematical formalization using Mathlib.

Chapters 17 and 18 provide practical engineering guidance and survey the ecosystem. Chapter 17 covers project organization, performance engineering, debugging and best practices. Chapter 18 surveys the Lean ecosystem, version history, community and future directions.

Throughout the book, code examples are complete and self-contained. They are designed to be runnable in a Lean project, not just illustrative snippets. Every major concept is accompanied by working examples that show not just the syntax but also the elaboration, type checking, or proof process. Where design decisions matter, the book explains the trade-offs. Where there are common pitfalls, the book identifies them and shows how to avoid them.

One important note about scope: this book treats Lean as a single system that is both a programming language and a proof assistant. It does not present two separate tutorials concatenated together. The programming and proving aspects are unified in Lean’s design and this book reflects that unity. When discussing proofs, the book explains the underlying program terms. When discussing programming, the book explains the type-theoretic foundation that makes verification possible.

The book distinguishes carefully between three layers of the Lean ecosystem. Lean core is the language and its primitive constructs: the type theory, the syntax, the built-in tactics and commands and the standard library that ships with Lean itself. The Lean standard library provides foundational types, basic operations, collections and common utilities. Mathlib is a separate, community-developed library that builds on Lean core to formalize large areas of mathematics. Much of the functionality people associate with “Lean” actually lives in Mathlib, not in Lean core. The book identifies where functionality comes from and does not conflate these layers.

Where features are version-specific, experimental, or deprecated, the book states so explicitly. Lean is an evolving system and what is true in one version may not be true in another. The book targets Lean 4 in the 4.33.0 to 4.35.0

range, which represents the current stable and development versions as of 2026.

If you want to follow along as you read, you should install Lean 4 and set up a development environment. Chapter 2 explains how to do this. The VS Code extension with the Lean language server is the recommended setup for most users.

Now let us begin.

Chapter 2: Getting Started with Lean 4

Installing Lean and Lake

Setting up a Lean 4 development environment involves three components: the Lean compiler itself, Lake the build system and package manager and a text editor with language server support. The recommended installation method uses elan, a toolchain manager similar to rustup that installs and manages Lean versions.

The official installation process starts at <https://lean-lang.org/install>. The simplest path is through Visual Studio Code. First, install VS Code from <https://code.visualstudio.com>. Then install the Lean 4 extension from the VS Code marketplace. Once installed, open the command palette and run the Lean setup command, or open the setup guide directly via the extension. The extension will guide you through installing elan and the Lean toolchain.

If you prefer a manual installation, the process is straightforward. On most Unix-like systems, run:

```
1 curl https://raw.githubusercontent.com/leanprover/elan/master/elan-init.sh  
   ↪ -sSf | sh
```

On Windows, download the elan installer from the Lean website. The installer places elan, lake and lean on your system path. You can verify the installation by running:

```
1 lean --version  
2 lake --version  
3 elan --version
```

These commands should display version numbers. The lean command is the Lean compiler. Lake is the build system and package manager, bundled with Lean 4 starting from the initial release. Elan manages toolchain versions and allows switching between Lean versions.

Lake replaced the separate lake tool used in early Lean 4 versions. It is now distributed as part of the Lean toolchain. Recent versions of Lake use a declarative TOML configuration file, `lakefile.toml`, by default; the earlier `lakefile.lean` format, written in a Lean domain-specific language, is still supported.

Your First Lean Project

Create a new Lean project using Lake. Lake initializes the current directory, so create the directory first:

```
1 mkdir hello_lean
2 cd hello_lean
3 lake init
```

The `lake init` command creates the following files in the current directory:

```
1 hello_lean/
2   lakefile.toml
3   lake-manifest.json
4   lean-toolchain
5   Main.lean
6   HelloLean.lean
7   HelloLean/
8     Basic.lean
9   README.md
```

The file `lakefile.toml` is the project configuration. A minimal example, exactly as generated by `lake init`, looks like this:

```
1 name = "hello_lean"
2 version = "0.1.0"
3 defaultTargets = ["hello_lean"]
4
5 [[lean_lib]]
6 name = "HelloLean"
7
8 [[lean_exe]]
9 name = "hello_lean"
10 root = "Main"
```

This declares a package named `hello_lean` with version 0.1.0 and one default build target: an executable whose entry module is `Main`. The `lean-toolchain` file pins the Lean version (for example, `v4.34.1`). To depend on `Mathlib4`, add a `require` entry:

```
1  [[require]]
2  name = "mathlib"
3  scope = "leanprover-community"
4  git = "https://github.com/leanprover-community/mathlib4"
5  rev = "v4.34.1"
```

The `lake-manifest.json` file is generated by Lake and tracks the exact versions of dependencies for reproducible builds. Do not edit it manually.

The file `Main.lean` is the starter source file for the executable. Open it and you will see something like:

```
1  import HelloLean
2
3  def main : IO Unit :=
4    IO.println s!"Hello, {hello}!"
```

This is a complete Lean program. The `def main` declaration defines the entry point of an executable. It returns an IO action that prints a string; the `s!` macro string-interpolates the value of `hello`, which `HelloLean/Basic.lean` defines as “world”. To build and run this project:

```
1  lake build
2  ./lake/build/bin/hello_lean
```

The output is:

```
1  Hello, world!
```

Lake compiles the project and places the executable in `.lake/build/bin`. You can also run code directly without building an executable using the `#eval` command. Add this line to your Lean file:

```
1 #eval "Hello, Lean 4!"
```

When you save the file in VS Code with the Lean extension installed, you will see the output in the Lean output panel.

The Lean ecosystem distinguishes between three kinds of Lean files and packages. Libraries are packages meant to be imported by other packages. Executables are packages that produce runnable programs. Both use the same language and tooling. Test files are Lean files that use assertions and checks to verify properties of the code. The same Lake project can contain all three kinds of targets.

Editors, LSP and the Lean Environment

Lean is designed for interactive use. You write code and proofs incrementally and the Lean language server continuously type-checks your work, shows errors, displays proof states and provides completions. This interactive feedback is essential for effective Lean development, especially for theorem proving.

The recommended editor is Visual Studio Code with the Lean 4 extension. The extension provides:

- Real-time type checking and error reporting
- Proof state display in tactic mode
- Hover information showing types and definitions
- Go-to-definition navigation
- Code completion and documentation
- Quick fixes for common errors
- A command palette for Lean-specific operations
- Output panels for build messages and evaluation results

Open a Lean file in VS Code and the extension activates the Lean language server automatically. The language server runs in the background, processing your file as you edit it. It type-checks each declaration and tactic step and reports problems in the Problems panel and inline in the editor.

When writing proofs in tactic mode, the proof state is displayed in a dedicated view. The proof state shows the current goal: what proposition you are trying to prove, what hypotheses are available and what local variables are

in scope. This feedback is crucial for developing proofs interactively. Each tactic transforms the proof state and you can see immediately whether your tactic made progress or failed.

Other editors support Lean through the Language Server Protocol. Neovim users can use the Lean extension through `nvim-lspconfig`. Emacs users can use `lean-mode` with `eglot` or `lsp-mode`. However, VS Code currently offers the most complete and polished Lean experience.

The Lean language server is itself written in Lean and runs as a separate process from the editor. It communicates with the editor via LSP messages over standard I/O or a socket. The server maintains an in-memory representation of all loaded files and their dependencies, incrementally checking changes as you edit. This architecture allows fast response times even for large projects.

A key feature of the Lean language server is the info tree. As Lean elaborates your code, it builds a tree of metadata: types of subterms, intermediate proof states, elaboration steps and error messages. The language server uses this info tree to provide rich interactive feedback: hover tooltips show types at arbitrary positions, the proof state view shows the state after each tactic and the “Locate error origin” feature traces errors back to their source. The info tree is stored in `.ilean` files and can be inspected programmatically.

Running and Interacting with Lean

Lean provides several mechanisms for running code and inspecting terms directly in source files, using commands that start with the hash symbol. These commands are evaluated during elaboration and their output is shown in the editor or terminal.

The `#eval` command runs a term: it is compiled to machine code and the result is printed as part of the build or elaboration. The term must have a type that Lean knows how to print, typically a basic type like `Nat`, `Bool`, `String`, or a type with a `Repr` instance:

```

1 #eval 2 + 3
2 -- Output: 5
3
4 #eval List.range 5
5 -- Output: [0, 1, 2, 3, 4]
6
7 #eval "Hello" ++ " " ++ "World"
8 -- Output: "Hello World"

```

The `#eval` command actually generates and executes code. It compiles the term to C, compiles the C code with a system compiler, links it with the Lean runtime and runs the resulting executable. This process can take a moment for complex terms and the output includes the result as printed by the Repr instance.

The `#reduce` command normalizes a term using Lean's reduction rules without generating executable code. It applies beta-reduction for function application, delta-reduction for unfolding definitions, iota-reduction for inductive type elimination and other computational rules. This is useful for checking that a term computes to the expected value without the overhead of code generation:

```

1 def double (n : Nat) := n + n
2
3 #reduce double 3
4 -- Output: 6
5
6 #reduce List.map double [1, 2, 3]
7 -- Output: [2, 4, 6]

```

The distinction between `#eval` and `#reduce` matters for terms that involve side effects or non-logical computation. The `#eval` command runs actual code and can perform I/O, while `#reduce` only applies reduction rules defined by the type theory.

The `#check` command prints the type of a term:

```

1 #check Nat.add
2 -- Output: Nat.add : Nat → Nat → Nat
3
4 #check List.map
5 -- Output: List.map.{u_1, u_2} {α : Type u_1} {β : Type u_2} (f : α → β) (l :
  ↳ List α) : List β

```

This is extremely useful for learning the API: you can check what type a definition has without reading its declaration. Note that `#check` shows implicit arguments (the parameters in curly braces), which are inferred by the elaborator when the function is used.

The `#print` command displays the definition of a constant:

```

1 #print Nat.add
2 -- Output shows the definition of addition

```

You can also print axioms, theorems and equational lemmas generated by recursive definitions. This is useful for debugging: if a proof is not going the way you expect, you can print relevant definitions to understand how they expand.

The `#synth` command attempts to synthesize a type class instance:

```

1 #synth Add Nat
2 -- Output: instAddNat
3
4 #synth Monad IO
5 -- Output: instMonadEIO

```

This is useful for understanding how type class resolution works and for debugging instance resolution failures.

The `#guard_msgs` command is a testing facility that asserts expected compiler messages. It is used in test files to verify that specific errors or warnings are produced. The expected messages are listed in the docstring directly above the command, one per line with their severity prefix; the file fails to build if the actual messages differ:

```
1 /--  
2 info: Nat.succ (n : Nat) : Nat  
3 -/  
4 #guard_msgs in  
5 #check Nat.succ
```

This is primarily useful for library maintainers who want to test that error messages and behaviors are as expected.

For more serious interactive exploration, the Lean REPL (Read-Eval-Print Loop) is available as a community project at <https://github.com/leanprover-community/repl>. It provides an interactive terminal interface similar to Python's REPL or Node.js, where you can enter terms and see their types and values immediately.

Lean 4Web is another community tool that runs Lean in the browser, allowing you to write and execute Lean code without any local installation. It is useful for quick experiments and teaching demonstrations.

The Lake Build System and Package Manager

Lake is Lean's build system and package manager. It handles project configuration, dependency management, compilation, caching and distribution. Lake is designed to integrate seamlessly with the Lean toolchain and to support large-scale developments.

By default, Lake projects use `lakefile.toml`, a declarative TOML configuration; the `lakefile.lean` format, written in a Lean domain-specific language (DSL), is still supported and allows computed configuration. A typical `lakefile.lean` configuration includes:

```

1  import Lake
2  open Lake DSL
3
4  package "my_project" where
5    -- Package name (must match directory name)
6    -- Version using SemVer notation
7    version := v!"4.33.0"
8
9    -- Description
10   description := "A sample Lean 4 project"
11
12   -- Options passed to the Lean compiler (here: enable Unicode function syntax)
13   leanOptions := #[
14     (\pp.unicode.fun, true)
15   ]
16
17   -- Dependencies: require other packages
18   require mathlib from git
19     "https://github.com/leanprover-community/mathlib4" @ "v4.34.1"
20
21   -- Library target: compiled package meant for importing
22   @[default_target]
23   lean_lib MyProject
24
25   -- Executable target: compiled program
26   lean_exe "my_exe" where
27     root := `MyExe

```

The version syntax `v!"4.33.0"` is a Lake macro that parses a string into a semantic version value. Dependencies are specified with `require`, which takes a name, a source (git, path, or online registry) and a version or commit. The `lean_lib` target specifies that this package produces a library meant to be imported by other packages. The `lean_exe` target specifies that it produces a runnable program with a designated entry module, written as a backtick name literal (`MyExe`). The `@[default_target]` attribute marks the target that bare lake build compiles by default.

Key Lake commands:

- `lake init`: initialize the current directory as a new project
- `lake build`: build all default targets in the project
- `lake build target_name`: build a specific target
- `lake clean`: remove build artifacts
- `lake update`: update dependencies to their latest pinned versions
- `lake upgrade target`: upgrade a specific dependency

- `lake script run target args`: run a script target with arguments
- `lake exe`: run the default executable

Lake uses content-addressed caching: build outputs are stored based on the hash of their inputs. This means that if a source file has not changed, Lake skips recompiling it, even across different machines. The cache is stored in the `.lake` directory within the project.

For reproducibility, Lake maintains a `lake-manifest.json` file that records the exact versions of all dependencies. When you clone a Lean project, Lake reads this manifest and installs the exact dependency versions used by the original developer. This ensures that the project builds and behaves the same way on different machines.

Lake also supports custom targets and scripts. You can define custom build steps, test targets and documentation generation targets in your `lakefile.lean`. A script target is a function of the arguments (a list of strings) returning an exit code:

```
1  -- A custom script target
2  script my_script (args) do
3    IO.println s!"Running with {args}"
4    return 0
```

Run it with `lake script run my_script arg1 arg2`.

Lake handles the meta phase of compilation automatically. Lean distinguishes between object-level code (code that runs at program runtime) and meta-level code (code that runs during elaboration, such as tactics and macros). Lake tracks which files contain meta code and ensures they are compiled and loaded during the meta phase, before regular compilation begins. You mark a file as containing meta code by adding `meta` to the module header or by importing it with `meta import`.

Project Structure and Configuration

Organizing a Lean project well is important for maintainability, especially for large developments. A typical project structure for a medium to large project looks like this:

```

1 my_project/
2   lakefile.toml
3   lake-manifest.json
4   lean-toolchain
5   MyProject.lean      -- Main library module
6   MyProject/
7     Arithmetic.lean   -- Arithmetic definitions and proofs
8     DataStructures.lean -- Data structure definitions
9     Tactics.lean      -- Custom tactics
10    Utils.lean        -- Utility definitions
11  examples/
12    example1.lean      -- Example usage
13  tests/
14    test_arithmetic.lean -- Tests
15    test_data_structures.lean

```

By default, Lake looks for source files in the project root directory (the earlier convention used a `src/` directory, which can be restored by setting `sourceDir` in the lakefile). Each `.lean` file corresponds to a module and the module name is derived from the file path relative to the source directory. For example, `MyProject/Arithmetic.lean` defines the module `MyProject.Arithmetic`.

The `examples` directory contains standalone files demonstrating how to use the library. The `tests` directory contains test files that verify properties of the code. Lake can be configured to build and run these files as part of the build process.

Within each `.lean` file, the module name is not declared explicitly: Lean derives it from the file path. This file, `MyProject/Arithmetic.lean`, is the module `MyProject.Arithmetic`.

Use imports to depend on other modules:

```

1 import MyProject.DataStructures
2 import Mathlib.Data.List.Basic

```

Use namespaces to group related declarations:

```

1 namespace MyProject.Arithmetic
2
3 def add (a b : Nat) := a + b
4 def mul (a b : Nat) := a * b
5
6 end MyProject.Arithmetic

```

The namespace command establishes a naming context. All declarations within the namespace are prefixed with the namespace name when referred to from outside. You can open a namespace to avoid typing the full prefix:

```

1 namespace MyProject.Arithmetic
2 def add (a b : Nat) := a + b
3 end MyProject.Arithmetic
4
5 open MyProject.Arithmetic
6
7 -- Now you can write add instead of MyProject.Arithmetic.add
8 #eval add 1 2

```

Use sections to manage local variables and assumptions:

```

1 section
2 variable (n m : Nat)
3
4 theorem add_comm : n + m = m + n := by
5   induction n with
6   | zero => simp
7   | succ n' ih =>
8     rw [Nat.succ_add, ih, Nat.add_succ]
9
10 end

```

The variable declaration makes n and m available as implicit parameters to all subsequent definitions and as hypotheses to all subsequent theorems within the section. This reduces boilerplate and improves readability.

For large projects, namespace organization is critical. Mathlib uses a hierarchical namespace structure that reflects the mathematical content: `Mathlib.Algebra.Group.Basic` for basic group theory, `Mathlib.Topology.MetricSpace.Basic` for metric spaces and so on. This structure allows precise imports and avoids namespace pollution.

Lean’s module system provides encapsulation. Declarations are public by default and visible to every module that imports the defining file. To restrict a declaration to the defining file, mark it private:

```
1 namespace MyProject.API
2
3 -- Private: accessible only within this module (file)
4 private def internal_helper (n : Nat) := n + 1
5
6 -- Public by default: visible to importers of this module
7 def public_function (n : Nat) := internal_helper n
8
9 end MyProject.API
10
11 #eval MyProject.API.public_function 41
```

When another module imports `MyProject.API`, it can use `public_function`, but it cannot refer to `internal_helper` directly. This encapsulation is useful for maintaining clean APIs in large projects.

Lake also supports publishing packages to registries and distributing them via content-addressable archives (.ltar files). The lake upload command publishes a package to the registry and other users can depend on it by name and version. This ecosystem of packages makes it easy to reuse code across projects.

Getting started with Lean is about setting up the tools and understanding the basic workflows: creating projects, writing code, building, running and using the interactive feedback from the language server. The rest of this book explores the language itself in detail.

Chapter 3: The Lean Syntax and Basic Types

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/lean4programmingproofandformalverification>.

Modules, Namespaces and Imports

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/lean4programmingproofandformalverification>.

Identifiers and Declarations

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/lean4programmingproofandformalverification>.

Basic Types and Literals

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/lean4programmingproofandformalverification>.

Variables and Local Definitions

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/lean4programmingproofandformalverification>.

Functions and Applications

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/lean4programmingproofandformalverification>.

Pattern Matching

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/lean4programmingproofandformalverification>.

Chapter 4: Dependent Types and the Foundations

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/lean4programmingproofandformalverification>.

From Simple Types to Dependent Types

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/lean4programmingproofandformalverification>.

The Calculus of Constructions

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/lean4programmingproofandformalverification>.

Universes and Polymorphism

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/lean4programmingproofandformalverification>.

Pi Types and Dependent Functions

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/lean4programmingproofandformalverification>.

Dependent Pairs and Sigma Types

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/lean4programmingproofandformalverification>.

Definitional Equality and Conversion

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/lean4programmingproofandformalverification>.

Chapter 5: Inductive Types and Data Structures

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/lean4programmingproofandformalverification>.

Defining Inductive Types

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/lean4programmingproofandformalverification>.

Pattern Matching and Recursion

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/lean4programmingproofandformalverification>.

Lists, Vectors and Indexed Types

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/lean4programmingproofandformalverification>.

Inductive Families

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/lean4programmingproofandformalverification>.

Algebraic Data Types in Practice

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/lean4programmingproofandformalverification>.

Recursive and Well-founded Functions

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/lean4programmingproofandformalverification>.

Chapter 6: Functional Programming in Lean

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/lean4programmingproofandformalverification>.

Higher-order Functions and Composition

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/lean4programmingproofandformalverification>.

Type Classes and Generic Programming

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/lean4programmingproofandformalverification>.

Structures and Records

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/lean4programmingproofandformalverification>.

Monads and Do Notation

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/lean4programmingproofandformalverification>.

Standard Library Collections

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/lean4programmingproofandformalverification>.

Performance and Mutation

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/lean4programmingproofandformalverification>.

Chapter 7: Propositions as Types

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/lean4programmingproofandformalverification>.

Prop and Type: Two Universes of Discourse

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/lean4programmingproofandformalverification>.

Logical Connectives as Types

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/lean4programmingproofandformalverification>.

Quantifiers as Dependent Types

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/lean4programmingproofandformalverification>.

Proof Terms and Proof Checking

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/lean4programmingproofandformalverification>.

Classical and Constructive Logic

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/lean4programmingproofandformalverification>.

Decidability and Computation

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/lean4programmingproofandformalverification>.

Chapter 8: Writing Proofs in Lean

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/lean4programmingproofandformalverification>.

Term-Style Proofs

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/lean4programmingproofandformalverification>.

Tactic Mode Basics

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/lean4programmingproofandformalverification>.

Core Tactics: intro, apply, exact, refine

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/lean4programmingproofandformalverification>.

Breaking Down Goals: constructor, cases, induction

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/lean4programmingproofandformalverification>.

Structuring Proofs: have, show, calc

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/lean4programmingproofandformalverification>.

When to Use Tactics Versus Terms

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/lean4programmingproofandformalverification>.

Chapter 9: Equality and Rewriting

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/lean4programmingproofandformalverification>.

Definitional and Propositional Equality

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/lean4programmingproofandformalverification>.

The Equality Type and Its Eliminator

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/lean4programmingproofandformalverification>.

The `rw` and `subst` Tactics

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/lean4programmingproofandformalverification>.

Calc Blocks for Chain Reasoning

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/lean4programmingproofandformalverification>.

Function Extensionality and Congruence

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/lean4programmingproofandformalverification>.

Quotients and Equivalence Relations

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/lean4programmingproofandformalverification>.

Chapter 10: Automation and Simplification

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/lean4programmingproofandformalverification>.

The Simplifier: `simp` and `simp_lemmas`

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/lean4programmingproofandformalverification>.

Rewrite Tactics and Strategies

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/lean4programmingproofandformalverification>.

Decision Procedures: `decide` and `native_decide`

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/lean4programmingproofandformalverification>.

Arithmetic Automation

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/lean4programmingproofandformalverification>.

Proof Search: `aesop` and Other Tools

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/lean4programmingproofandformalverification>.

When Automation Helps and Hurts

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/lean4programmingproofandformalverification>.

Chapter 11: The Lean Type System

Deep Dive

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/lean4programmingproofandformalverification>.

Lean's Type Theory: A Formal View

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/lean4programmingproofandformalverification>.

Inductive Type Recursors and Elimimators

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/lean4programmingproofandformalverification>.

Computation Rules and Reduction

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/lean4programmingproofandformalverification>.

Termination and Well-founded Recursion

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/lean4programmingproofandformalverification>.

Universe Levels and Size Constraints

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/lean4programmingproofandformalverification>.

Type Checking and Soundness

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/lean4programmingproofandformalverification>.

Chapter 12: Implicit Arguments and Type Class Resolution

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/lean4programmingproofandformalverification>.

Implicit Arguments and Inference

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/lean4programmingproofandformalverification>.

Type Classes: Concept and Mechanism

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/lean4programmingproofandformalverification>.

Instance Resolution and Search

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/lean4programmingproofandformalverification>.

Coercions and Canonical Constructions

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/lean4programmingproofandformalverification>.

Type Class Hierarchies and Algebra

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/lean4programmingproofandformalverification>.

Debugging Type Class Failures

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/lean4programmingproofandformalverification>.

Chapter 13: The Lean Architecture

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/lean4programmingproofandformalverification>.

The Trusted Computing Base

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/lean4programmingproofandformalverification>.

From Source to Proof Term: The Elaboration Pipeline

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/lean4programmingproofandformalverification>.

The Parser and Syntax Trees

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/lean4programmingproofandformalverification>.

The Kernel: Proof Checking

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/lean4programmingproofandformalverification>.

The Compiler and Runtime

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/lean4programmingproofandformalverification>.

Untrusted but Useful: Automation and Tooling

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/lean4programmingproofandformalverification>.

Chapter 14: Metaprogramming and Extensibility

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/lean4programmingproofandformalverification>.

Why Extend Lean?

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/lean4programmingproofandformalverification>.

Syntax Extensions and Custom Notation

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/lean4programmingproofandformalverification>.

Hygienic Macros

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/lean4programmingproofandformalverification>.

Elaborators and Commands

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/lean4programmingproofandformalverification>.

Tactic Implementation

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/lean4programmingproofandformalverification>.

Building Domain-Specific Languages

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/lean4programmingproofandformalverification>.

Chapter 15: Formal Verification and Verified Programming

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/lean4programmingproofandformalverification>.

Specification: Pre-, Post- and Invariants

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/lean4programmingproofandformalverification>.

Verifying Algorithms and Data Structures

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/lean4programmingproofandformalverification>.

Functional Correctness and Properties

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/lean4programmingproofandformalverification>.

Verified Compilers and Translations

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/lean4programmingproofandformalverification>.

Cryptographic and Security Properties

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/lean4programmingproofandformalverification>.

Case Study: A Verified Data Structure

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/lean4programmingproofandformalverification>.

Chapter 16: Mathematical Formalization with Mathlib

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/lean4programmingproofandformalverification>.

Lean Core, Stdlib and Mathlib

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/lean4programmingproofandformalverification>.

Working with Existing Formalizations

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/lean4programmingproofandformalverification>.

Algebra and Type Class Hierarchies

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/lean4programmingproofandformalverification>.

Analysis and Topology

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/lean4programmingproofandformalverification>.

Discrete Mathematics and Combinatorics

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/lean4programmingproofandformalverification>.

Engineering Large Formalizations

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/lean4programmingproofandformalverification>.

Chapter 17: Practical Engineering and Best Practices

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/lean4programmingproofandformalverification>.

Project Organization at Scale

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/lean4programmingproofandformalverification>.

Namespace and Module Design

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/lean4programmingproofandformalverification>.

Performance Engineering

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/lean4programmingproofandformalverification>.

Documentation and Readability

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/lean4programmingproofandformalverification>.

Testing and Debugging Proofs

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/lean4programmingproofandformalverification>.

Common Pitfalls and Solutions

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/lean4programmingproofandformalverification>.

Chapter 18: The Lean Ecosystem and Beyond

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/lean4programmingproofandformalverification>.

Lean 4 Version History and Releases

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/lean4programmingproofandformalverification>.

Tooling and Integrations

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/lean4programmingproofandformalverification>.

Community and Contribution

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/lean4programmingproofandformalverification>.

Interoperability and FFIs

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/lean4programmingproofandformalverification>.

Real-World Deployments

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/lean4programmingproofandformalverification>.

Where Lean Is Headed

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/lean4programmingproofandformalverification>.

Conclusion

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/lean4programmingproofandformalverification>.

References

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/lean4programmingproofandformalverification>.