Functional Programming in Lean

Lecture 1: Introduction

Marius Weidner

Albert-Ludwigs-Universität Freiburg, Germany

Winter Semester 2026/27

Coordinates

Staff



Prof. Dr. Peter Thiemann
lecturer
thiemann@informatik.uni-freiburg.de



Marius Weidner
assistant
weidner@cs.uni-freiburg.de

Lecture

Tuesday, 14:00–16:00
Building 106, Room 04 007

Tutorial

Monday, 10:00–12:00
Building 101, Room 00-017/19

Links

Website: https://proglang.github.io/teaching/26ws/lean.html

Forum: https://chat.proglang.de/join/dzmad3m76sci2jy3yoetsfwb/

Administrivia

Lecture: 90 minutes per week

- short slide decks, **live programming** in Lean

Tutorial: 90 minutes per week

- one exercise sheet per week, published on the website
- sheets are **not handed in and not graded**
- discussed in the next tutorial session

Final exam

- in the computer pools, closed book, **live programming**
- date and time: tba

Contents of the course

Note

Only the draft plan for the semester. Topics and order may still change.

Core (weeks 1–13)

1. Functions and types
2. Inductive data types
3. Lists, higher-order functions
4. Recursion and evaluation
5. Type classes
6. Error handling and `IO`
7. Parser combinators
8. A monadic interpreter
9. Proving program properties
10. Lambda calculus
11. Indexed types, typed evaluators
12. Macros and tactics
13. Specifications, code, proofs

Bonus (weeks 14–16)

- Type inference
- Monad transformers
- Functors and categories

Goal

Read, write, and reason about functional programs.

Reading and tools

- **Main text:** *Functional Programming in Lean*, David Thrane Christiansen
  https://lean-lang.org/functional_programming_in_lean/
- **Reference:** *Lean Language Reference*
  https://lean-lang.org/doc/reference/latest/
- **Theorem proving:** *Theorem Proving in Lean 4*
  https://lean-lang.org/theorem_proving_in_lean4/
- **Community:** Lean Zulip chat https://leanprover.zulipchat.com/
- **Tools:** Lean 4 and VS Code with the Lean 4 extension

What is Functional Programming?

A different approach to programming

Functions and values

rather than

Assignments and pointers

It will make you a better programmer

What is Functional Programming?

A different approach to programming

Functions and values

rather than

Assignments and pointers

It will make you a better programmer

What is Lean?

- A **functional programming language** *and* an **interactive theorem prover**
- Started in 2013 by Leonardo de Moura at Microsoft Research; Lean 4 is developed by the Lean FRO (Focused Research Organization) and an open-source community
- Based on dependent type theory: **programs and proofs live in one language**
- Lean has a self-hosted compiler: it is written in Lean and compiles to C, and from there to efficient native code
- Home of **Mathlib**, a library of more than two million lines of formalized mathematics
- Used in industry for verified software, and by mathematicians to check proofs

Why Lean?

- Lean is a very high-level language

  (many details taken care of automatically)
- Lean is expressive and concise

  (can achieve a lot with little effort)
- Lean is good at handling complex data and combining components
- Lean has a precise type system that catches mistakes before running the program
- Lean lets you **state and prove** that your program is correct

  (in the same file, checked by the same tool)
- Lean has excellent interactive tooling

  (the infoview shows types, values, and goals as you type)

Functional vs Imperative Programming: Variables

Functional (Lean)

```
def x : Nat := 5
```

Variable x has value 5 forever

Imperative (Rust)

```
let mut x = 5;
...
x = x + 1;
```

Variable x can change its content over time

But you can *shadow* a variable

```
def y : Unit :=
  let x := 5
  let x := x + 1   -- a new x that hides the old one
  ()
```

Functional vs Imperative Programming: Variables

Functional (Lean)

```
def x : Nat := 5
```

Variable x has value 5 forever

Imperative (Rust)

```
let mut x = 5;
...
x = x + 1;
```

Variable x can change its content over time

But you can *shadow* a variable

```
def y : Unit :=
  let x := 5
  let x := x + 1   -- a new x that hides the old one
  ()
```

Functional vs Imperative Programming: Variables

Functional (Lean)

```
def x : Nat := 5
```

Variable x has value 5 forever

Imperative (Rust)

```
let mut x = 5;
...
x = x + 1;
```

Variable x can change its content over time

But you can *shadow* a variable

```
def y : Unit :=
  let x := 5
  let x := x + 1   -- a new x that hides the old one
  ()
```

Functional vs Imperative Programming: Functions

Functional (Lean)

```
def f (x y : Nat) : Nat := 2 * x + y
#eval f 42 16   -- always 100
```

No side effects: the result depends **only** on the inputs

Imperative (Rust)

```
fn f(x: i32) -> i32 {
    let input = io::read_to_string(io::stdin()).unwrap();
    let y: i32 = input.trim().parse().unwrap();
    2 * x + y
}
let z = f(42);   // who knows?
```

Side effect: `f` reads `y` from the input. Its type does not say so

Functional vs Imperative Programming: Functions

Functional (Lean)

```
def f (x y : Nat) : Nat := 2 * x + y
#eval f 42 16   -- always 100
```

No side effects: the result depends **only** on the inputs

Imperative (Rust)

```
fn f(x: i32) -> i32 {
    let input = io::read_to_string(io::stdin()).unwrap();
    let y: i32 = input.trim().parse().unwrap();
    2 * x + y
}
let z = f(42);   // who knows?
```

Side effect: `f` reads `y` from the input. Its type does not say so

Equational Reasoning and Referential Transparency

Lean

1. Variables have a fixed value
2. Functions have a fixed meaning that does not depend on context
   (Actually, the same as item 1!)

Consequence: Equational Reasoning

- Every (sub-)expression in a program can be replaced with its value, and
- every function call can be replaced with its definition
- … without changing the meaning of the program!

In Lean, this reasoning can be *checked by the compiler*

```
theorem f_zero (y : Nat) : f 0 y = y := by simp [f]
```

Equational Reasoning and Referential Transparency

Lean

1. Variables have a fixed value
2. Functions have a fixed meaning that does not depend on context
   (Actually, the same as item 1!)

Consequence: Equational Reasoning

- Every (sub-)expression in a program can be replaced with its value, and
- every function call can be replaced with its definition
- … without changing the meaning of the program!

In Lean, this reasoning can be *checked by the compiler*

```
theorem f_zero (y : Nat) : f 0 y = y := by simp [f]
```

Equational Reasoning and Referential Transparency

Lean

1. Variables have a fixed value
2. Functions have a fixed meaning that does not depend on context
   (Actually, the same as item 1!)

Consequence: Equational Reasoning

- Every (sub-)expression in a program can be replaced with its value, and
- every function call can be replaced with its definition
- … without changing the meaning of the program!

In Lean, this reasoning can be *checked by the compiler*

```
theorem f_zero (y : Nat) : f 0 y = y := by simp [f]
```

Many more features that make programs more concise

- Static typing
- Type inference
- Inductive (algebraic) data types
- Pattern matching
- Polymorphic types
- Type classes
- Monads & do-notation (for `IO`, errors, state, …)
- Dependent types
- Theorems and tactics
- Metaprogramming (macros)
- …

Let's get started!

Learning goals for today

- Function definitions and type signatures
- **`#eval`** and **`#check`**
- Pure expressions: input types and result type, evaluation without side effects
- Reading a type error

Setting up Lean

1. Install `elan`, the Lean version manager:
   `https://lean-lang.org/install/`
2. Install VS Code and the *Lean 4* extension
3. Create a new file ending in `.lean` and open it in VS Code
4. Type a first line and check that Lean answers:

```
#eval 1 + 1
```

The infoview

Place the cursor on a line. The infoview panel shows messages, values, and types for that position.

A first definition

- A **definition** gives a name to a value or a function
- Definitions are put in a text file ending in `.lean`

```
def double (n : Nat) : Nat := n + n
```

- `def`: keyword that starts a definition
- `double`: name of the function (defined here)
- `(n : Nat)`: argument `n` of type `Nat` (natural numbers $0, 1, 2, \ldots$)
- `: Nat`: the type of the result
- `n + n`: expression that computes the result

Using the definition: #eval and #check

Evaluate an expression

```
#eval double 7
```

```
14
```

Ask for the type of an expression

```
#check double
#check double 7
```

```
double (n : Nat) : Nat
double 7 : Nat
```

- Function application is written by juxtaposition: `double 7`, no parentheses
- **#eval** and **#check** are *commands*, not part of the program

A conditional expression

```
def larger (x y : Nat) : Nat := if x < y then y else x
```

- Two arguments of the same type: `(x y : Nat)`
- **`if`** `...` **`then`** `...` **`else`** `...` is an **expression**: it has a value
- Both branches must have the same type (here `Nat`)
- There is no **`return`**: the body *is* the result

Your turn: predict before you evaluate

```
#eval larger 12 9
#eval larger 9 12
#eval larger (double 3) 5
```

A conditional expression

```
def larger (x y : Nat) : Nat := if x < y then y else x
```

- Two arguments of the same type: `(x y : Nat)`
- `if ... then ... else ...` is an **expression**: it has a value
- Both branches must have the same type (here `Nat`)
- There is no `return`: the body *is* the result

Your turn: predict before you evaluate

```
#eval larger 12 9
#eval larger 9 12
#eval larger (double 3) 5
```

Every value has a type

```
#check 7
#check "seven"
#check true
#check 7.5
#check double 7
```

```
7 : Nat
"seven" : String
Bool.true : Bool
7.5 : Float
double 7 : Nat
```

- `Bool.true` and `Bool.false` are the **constructors** of the type `Bool`
- Lean must know the type of every expression before it evaluates it

Reading a type error

```
#eval double "seven"
```

```
Application type mismatch: The argument
  "seven"
has type
  String
but is expected to have type
  Nat
in the application
  double "seven"
```

- **expected type**: what the function signature demands
- **actual type**: what the argument has
- The program is rejected *before* anything runs

Types matter

- Types determine how computations are performed
- A type annotation `(e : T)` specifies which type to use

```
#eval 7 - 9
#eval (7 - 9 : Int)
```

```
0
-2
```

```
#eval 123456789 * 123456789
#eval (123456789 * 123456789 : Float)
```

```
15241578750190521
15241578750190520.000000
```

- `Nat`: exact, but subtraction stops at 0 `Int`: exact, negative numbers `Float`: double precision, rounding

Type inference

- Lean **infers** (works out) the type of every expression
- Numeric literals default to `Nat` unless the context says otherwise
- Checks that all types match, before running the program

```
def price := 79          -- no type given
def answer := double 21  -- result type of double

#check price
#check answer
```

```
price : Nat
answer : Nat
```

Convention

Write the types of top-level definitions anyway. They document the code and give better error messages.

Anonymous functions

- Functions are values. They need not have a name.
- `fun n : Nat => n + 1` is the function that adds one to its argument

```
#check fun n : Nat => n + 1
#eval (fun n : Nat => n + 1) 41
```

```
fun n => n + 1 : Nat → Nat
42
```

- `Nat → Nat`: the type of functions from `Nat` to `Nat`
- The arrow is typed as `\to` or `->` in VS Code
- `def double (n : Nat) : Nat := n + n` is short for
  `def double : Nat → Nat := fun n => n + n`

A first theorem (preview)

- Equational reasoning is built into Lean
- A **`theorem`** states a proposition; the part after `:=` proves it

```
theorem double_zero : double 0 = 0 := by
  rfl
```

- **`by`** `rfl`: both sides evaluate to the same value
- Lean checks the proof like it checks a type: no output means success
- Much more on proofs later in the course

Live Coding!

Check your understanding

1. What is the type of `fun n : Nat => n + 1`?
2. What is the type of `larger 3`?
3. What does `#eval larger 3` print?
4. Why does `#eval 7 - 9` print `0`?

Hint for 3

```
could not synthesize a `Repr` or `ToString` instance for type
  Nat → Nat
```

A function has no printable representation. Its type is still fine.

Check your understanding

1. What is the type of `fun n : Nat => n + 1`?
2. What is the type of `larger 3`?
3. What does `#eval larger 3` print?
4. Why does `#eval 7 - 9` print `0`?

Hint for 3

```
could not synthesize a `Repr` or `ToString` instance for type
  Nat → Nat
```

A function has no printable representation. Its type is still fine.

Summary

- Functional programming: functions and values instead of assignments and pointers
- Lean: a functional language and a theorem prover in one
- `def name (arg : T) : R := body` defines a function
- **`#eval`** computes a value, **`#check`** shows a type
- Every expression has a type; mismatches are reported before running
- **`if`** `...` **`then`** `...` **`else`** and **`fun`** `x => ...` are expressions

Next week

Inductive data types: modelling data with constructors and pattern matching

