Topology Filters - Notes

Release 0.1

Carlos Caralps

Jul 16, 2021

CONTENTS

CHAPTER

ONE

INTRODUCTION

1.1 What is Lean?

Lean is an open source proof-checker and a proof-assistant. One can *explain* mathematical proofs to it and it can check their correctness. It also simplifies the proof writing process by providing `goals` and `tactics`.

Lean is built on top of a formal system called type theory. In type theory, the basic notions are "terms" and "types" — compare to "elements" and "sets" in set theory. Every term has a type, and types are just a special kind of term. Terms can be interpreted as mathematical objects, functions, propositions, or proofs. The only two things Lean can do is *create* terms and *check* their types. By iterating these two operations, we can teach Lean to verify complex mathematical proofs.

```
def x := 2 + 2                                     -- a natural number
def f (x : ℕ) := x + 3                             -- a function
def easy_theorem_statement := 2 + 2 = 4            -- a proposition
def fermats_last_theorem_statement                 -- another proposition
  :=
  ∀ n : ℕ,
  n > 2
  →
  ¬ (∃ x y z : ℕ, (x^n + y^n = z^n) ∧ (x ≠ 0) ∧ (y ≠ 0) ∧ (z ≠ 0))

theorem
easy_proof : easy_theorem_statement                -- proof of easy_theorem
:=
begin
  exact rfl,
end

theorem
my_hard_proof : fermats_last_theorem_statement        -- cheating!
:=
begin
  sorry,
end

#check x
#check f
#check easy_theorem_statement
#check fermats_last_theorem_statement
#check easy_proof
#check hard_proof
```

1.2 How to use these notes

Every once in a while, you will see a code snippet like this:

```
#eval "Hello, World!"
```

Clicking on the `try it!` button in the upper right corner will open a copy in a window so that you can edit it, and Lean provides feedback in the `Lean Infoview` window. We use this feature to provide exercises inline in the notes. We recommend attempting each exercise as you go along.

These notes are based a 5-day Lean crash course at Mathcamp 2020. We have adapted them to BIYSC 2021.

These notes provide a sneak-peek into the world of theorem proving in Lean and are by no means comprehensive. It is recommended that you simultaneously attempt the Natural Number Game. It is a fun (and highly addictive!) game that proves same basic properties of natural numbers in Lean.

1.3 Acknowledgments.

These notes are based on work of Apurva Nakade and Jalex Stark. Large chunks of these notes are taken directly from <https://apurvanakade.github.io/courses/lean_at_MC2020/>`__.

1.4 Useful Links.

1. Formalizing 100 theorems
2. Formalizing 100 theorems in Lean
3. **Articles, videos, blog posts, etc.**
    1. The Xena Project
    2. The Mechanization of Mathematics
    3. The Future of Mathematics
4. Lean Zulip chat group

CHAPTER TWO

GLOSSARY OF TACTICS

2.1 Implications in Lean

`exact`	If `P` is the target of the current goal and `hp` is a term of type `P`, then `exact hp,` will close the goal. Mathematically, this saying “this is *exactly* what we were required to prove”.
`intro`	If the target of the current goal is a function `P → Q`, then `intro hp,` will produce a hypothesis `hp : P` and change the target to `Q`. Mathematically, this is saying that in order to define a function from `P` to `Q`, we first need to choose an arbitrary element of `P`.

`have`	`have` is used to create intermediate variables. If `f` is a term of type `P → Q` and `hp` is a term of type `P`, then `have hq := f hp,` creates the hypothesis `hq : Q`.
`apply`	`apply` is used for backward reasoning. If the target of the current goal is `Q` and `f` is a term of type `P → Q`, then `apply f,` changes target to `P`. Mathematically, this is equivalent to saying “because `P` implies `Q`, to prove `Q` it suffices to prove `P`”.

2.2 Proof by contradiction

`ex-falso`	Changes the target of the current goal to `false`. The name derives from "ex falso, quodlibet" which translates to "from contradiction, anything". You should use this tactic when there are contradictory hypotheses present.
`by_cases`	If `P : Prop`, then `by_cases P,` creates two goals, the first with a hypothesis `hp: P` and second with a hypothesis `hp: ¬ P`. Mathematically, this is saying either `P` is true or `P` is false. `by_cases` is the most direct application of the law of excluded middle.
`by_contradiction`	If the target of the current goal is `Q`, then `by_contradiction,` changes the target to `false` and adds `hnq : ¬ Q` as a hypothesis. Mathematically, this is proof by contradiction.
`push_neg`	`push_neg,` simplifies negations in the target. For example, if the target of the current goal is `¬ ¬ P`, then `push_neg,` simplifies it to `P`. You can also push negations across a hypothesis `hp : P` using `push_neg at hp,`.
`contrapose!`	If the target of the current goal is `P → Q`, then `contrapose!,` changes the target to `¬ Q → ¬ P`. If the target of the current goal is `Q` and one of the hypotheses is `hp : P`, then `contrapose! hp,` changes the target to `¬ P` and changes the hypothesis to `hp : ¬ Q`. Mathematically, this is replacing the target by its contrapositive.

2.3 And / Or

`cases`	`cases` is a general tactic that breaks a complicated term into simpler ones. If `hpq` is a term of type `P ∧ Q`, then `cases hpq with hp hq,` breaks it into `hp : P` and `hp : Q`. If `hpq` is a term of type `P × Q`, then `cases hpq with hp hq,` breaks it into `hp : P` and `hp : Q`. If `fg` is a term of type `P ↔ Q`, then `cases fg with f g,` breaks it into `f : P → Q` and `g : Q → P`. If `hpq` is a term of type `P ∨ Q`, then `cases hpq with hp hq,` creates two goals and adds the hypotheses `hp : P` and `hq : Q` to one each.
`split`	`split` is a general tactic that breaks a complicated goal into simpler ones. If the target of the current goal is `P ∧ Q`, then `split,` breaks up the goal into two goals with targets `P` and `Q`. If the target of the current goal is `P × Q`, then `split,` breaks up the goal into two goals with targets `P` and `Q`. If the target of the current goal is `P ↔ Q`, then `split,` breaks up the goal into two goals with targets `P → Q` and `Q → P`.
`left`	If the target of the current goal is `P ∨ Q`, then `left,` changes the target to `P`.
`right`	If the target of the current goal is `P ∨ Q`, then `right,` changes the target to `Q`.

2.4 Quantifiers

`have`	If `hp` is a term of type `∀ x : X, P x` and `y` is a term of type `y` then `have hpy := hp(y)` creates a hypothesis `hpy : P y`.
`intro`	If the target of the current goal is `∀ x : X, P x`, then `intro x,` creates a hypothesis `x : X` and changes the target to `P x`.

`cases`	If `hp` is a term of type `∃ x : X, P x`, then `cases hp with x key,` breaks it into `x : X` and `key : P x`.
`use`	If the target of the current goal is `∃ x : X, P x` and `y` is a term of type `X`, then `use y,` changes the target to `P y` and tries to close the goal.

2.5 Proving "trivial" statements

`norm_num`	`norm_num` is Lean's calculator. If the target has a proof that involves *only* numbers and arithmetic operations, then `norm_num` will close this goal. If `hp : P` is an assumption then `norm_num at hp,` tries to use simplify `hp` using basic arithmetic operations.
`ring`	`ring,` is Lean's symbolic manipulator. If the target has a proof that involves *only* algebraic operations, then `ring,` will close the goal. If `hp : P` is an assumption then `ring at hp,` tries to use simplify `hp` using basic algebraic operations.
`linarith`	`linarith,` is Lean's inequality solver.
`simp`	`simp,` is a very complex tactic that tries to use theorems from the mathlib library to close the goal. You should only ever use `simp,` to *close a goal* because its behavior changes as more theorems get added to the library.

2.6 Equality

`refl`	If the current goal is of the form `X = X` or `P ↔ P`, then `refl` will finish the proof. As long as both sides are defined to be equal, this will work. For example, it will work with the goal `3 = 2 + 1` because *by definition* the number 3 is defined to be 2 plus one. Mathematically, this says "check that both sides are qual *by definition*".

`rw`	If `f` is a term of type `P = Q` (or `P ↔ Q`), then
	`rw f,` searches for `P` in the target and replaces it with `Q`.
	`rw ←f,` searches for `Q` in the target and replaces it with `P`.
	If additionally, `hr : R` is a hypothesis, then
	`rw f at hr,` searches for `P` in the expression `R` and replaces it with `Q`.
	`rw ←f at hr,` searches for `Q` in the expression `R` and replaces it with `P`.
	Mathematically, this is saying because `P = Q`, we can replace `P` with `Q` (or the other way around).