Tactics in Lean

Floris van Doorn

University of Pittsburgh

February 2, 2020

Mathlib contents

Subdirectory	LOC	Decls		Subdirectory	LOC	Decls
data	41849	10695		linear_algebra	4511	805
topology	17382	2709		computability	4205	575
tactic	12184	1679		group_theory	4191	1094
algebra	9830	2794		category	1770	389
analysis	7962	1237		number_theory	1394	228
order	6526	1542		logic	1195	403
category_theory	6299	1560		field_theory	1002	121
set_theory	6163	1394		geometry	848	70
measure_theory	6113	926		meta	784	135
ring_theory	5683	1080		algebraic_geometry	194	29

Total: 140k LOC (excluding whitespace/comments) and 34k declarations (as of December).

norm_num

Kevin Buzzard (27 Sep 2017) I want to make these sorts of calculations trivial:

```
example : (((3 : ℝ)/4)-12)<6  := sorry
example : (6 : ℝ) + 9 = 15    := sorry
example : (2 : ℝ) * 2 + 3 = 7 := sorry
example : (5 : ℝ) ≠ 8         := sorry
example : (6 : ℝ) < 10        := sorry
example : (7 : ℝ)/2 > 3       := sorry
```

norm_num

Mario Carneiro (2 Nov 2017) `norm_num` now solves all these goals.

```
example : ¬ (7-2)/(2*3) ≥ (1:ℝ) + 2/(3^2) := by norm_num
example : (6 : ℝ) + 9 = 15                  := by norm_num
example : (2 : ℝ)/4 + 4 = 3*3/2             := by norm_num
example : (((3 : ℝ)/4)-12)<6                := by norm_num
example : (5 : ℝ) ≠ 8                       := by norm_num
example : (10 : ℝ) > 7                      := by norm_num
example : (2 : ℝ) * 2 + 3 = 7               := by norm_num
example : (6 : ℝ) < 10                      := by norm_num
example : (7 : ℝ)/2 > 3                     := by norm_num

example : (1103 : ℤ) ≤ (2102 : ℤ)     := by norm_num
example : (110474 : ℤ) ≤ (210485 : ℤ) := by norm_num
example : (11047462383473829263 : ℤ) ≤ (21048574677772382462 : ℤ) :=
by norm_num
example : (210485742382937847263 : ℤ) ≤ (1104857462382937847262 : ℤ) :=
by norm_num
example : (210485987642382937847263 : ℕ) ≤ (11048512347462382937847262 : ℕ)
  := by norm_num
example : (210485987642382937847263 : ℚ) ≤ (11048512347462382937847262 : ℚ)
  := by norm_num
```

norm_num

Kevin Buzzard (11 Nov 2017) I love `norm_num`, I even use it to prove `0 < 1` nowadays. I use it to prove everything. It's perfect. Many thanks for `norm_num`.

norm_cast

Johan Commelin (13 Mar 2019) It's quite humiliating, but how do I kill:

```
example (p : ℕ) [p.prime] : (p : ℝ) > 1 := sorry
```

Paul-Nicolas Madelaine (9 Apr 2019) Here is the first version of the cast tactic I've been working on.

```
example (a : ℕ) (b : ℤ) : (a : ℚ) < b ↔ (a : ℝ) < b := by norm_cast
example (a b : ℤ)       : a = b ↔ (a : ℚ) = b        := by norm_cast
example (a b : ℕ)       : (a : ℤ) + b = (a + b : ℕ)  := by norm_cast
example (a : ℕ) (b : ℚ) : (a : ℂ) * b = ((a * b) : ℚ) := by norm_cast
example (a b : ℕ) : (((a : ℤ) : ℚ) : ℝ) + b = (a + (b : ℤ)) :=
by norm_cast
```

lift

Johan Commelin (9 Aug 2019) Suppose that `n : ℤ` and `h : n ≥ 0`. Then every mathematician (and especially if they are new to Lean) wants to say `n : ℕ`. But that is not possible.

Floris van Doorn (10 Apr 2019) PR'd the `lift` tactic.

```
example {P : ℤ → Prop} (n : ℤ) (hn : n ≥ 0) : P n :=
begin
  lift n to ℕ using hn,
  /- New goal:
  P : ℤ → Prop,
  n : ℕ
  ⊢ P ↑n -/
  sorry
end
```

Other examples

Other useful tactics that have been implemented:

- `library_search` searches the library to close the current goal.
- `suggest` searches the library for a lemma that is applicable.
- `simpa using h` closes the goal by simplifying both the goal and `h` to the same expression.
- `abel`, `ring`, `linarith`, `omega`: domain-specific automation.
- `tidy`, `finish`, `solve_by_elim`: general purpose automation .

rcases and rintro

`cases` destructs hypotheses, for example if `p : A × B` then `cases p with a b` gives two new hypothesis `a : A` and `b : B`.

`rcases` and `rintro` perform these operations recursively.
Before:

```
cases h with y y2, cases y2 with yS hy, cases yS with y0 yx,
```

After:

```
rcases h with ⟨y, ⟨y0, yx⟩, hy⟩,
```

Before:

```
intro p, cases p with p₁ p₂, cases p₁ with l hl, cases p₂ with u hu,
```

After:

```
rintro ⟨⟨l, hl⟩, ⟨u, hu⟩⟩,
```

simps

Before:

```
def yoneda C ⥤ (Cᵒᵖ ⥤ Type v₁) :=
{ obj := λ X,
  { obj := λ Y, unop Y ⟶ X,
    map := λ Y Y' f g, f.unop ≫ g,
    /- (two fields omitted for readability) -/ },
  map := λ X X' f, { app := λ Y g, g ≫ f } }

@[simp] lemma obj_obj (X : C) (Y : Cᵒᵖ) :
  (yoneda.obj X).obj Y = (unop Y ⟶ X) := rfl
@[simp] lemma obj_map (X : C) {Y Y' : Cᵒᵖ} (f : Y ⟶ Y') :
  (yoneda.obj X).map f = λ g, f.unop ≫ g := rfl
@[simp] lemma map_app {X X' : C} (f : X ⟶ X') (Y : Cᵒᵖ) :
  (yoneda.map f).app Y = λ g, g ≫ f := rfl
```

After:

```
@[simps] def yoneda : C ⥤ (Cᵒᵖ ⥤ Type v₁) :=
/- (definition is unchanged) -/
```

#lint

`#lint` is a semantic linter: it looks through the current file and looks for common mistakes in the declarations. Some mistakes that it catches:

- Have a hypothesis in a lemma that is never used;
- A declaration is incorrectly marked as a lemma or definition;
- A definition without documentation string.
- ...

localized notation

In Lean 3 notation is either local (to the current file or section) or global. You often want to use notation repeatedly, without it being global

```
localized "notation `ω` := ordinal.omega" in ordinal
```

You can get all notation in the `ordinal` locale by writing `open_locale ordinal`.

Writing Tactics: expr

We have reflection of expressions into Lean:

```
meta inductive expr (elaborated : bool := tt)
| var         {} : nat → expr
| sort        {} : level → expr
| const       {} : name → list level → expr
| mvar           : name → name → expr → expr
| local_const    : name → name → binder_info → expr → expr
| app            : expr → expr → expr
| lam            : name → binder_info → expr → expr → expr
| pi             : name → binder_info → expr → expr → expr
| elet           : name → expr → expr → expr → expr
| macro          : macro_def → list expr → expr
```

For example,

```
λ (x : ℕ), nat.add x x
```

is reflected as

```
(lam x default (const nat []) (app (app (const nat.add []) (var 0)) (var 0)))
```

Writing Tactics: tactic

The tactic monad allows us to define custom tactics:

```
meta def tactic := interaction_monad tactic_state
```

A tactic (`t : tactic` α) takes the current tactic state and runs a program to either

- succeed, and return the new tactic state and an element of α;
- fail with an error message.

There are hooks for tactics implemented in `C++`:

```
meta constant infer_type : expr → tactic expr
```

assumption

This allows us to write our own tactics.

```
/-- `find_same_type t es` tries to find in `es` an expression with type
   definitionally equal to `t` -/
meta def find_same_type : expr → list expr → tactic expr
| e []         := failed
| e (H :: Hs) :=
  do t ← infer_type H,
     (unify e t >> return H) <|> find_same_type e Hs

/-- `assumption` closes the goal if there is a hypothesis with the same type as
   the goal. -/
meta def assumption : tactic unit :=
do { ctx ← local_context,
     t   ← target,
     H   ← find_same_type t ctx,
     exact H }
<|> fail "assumption tactic failed"


example {p q : Prop} (h₁ : p ∨ q) (h₂ : q) : q := by assumption
```

Demo