# 06-ineq.lean

```
import Mathlib
```

Inequality

- Inequality `PartialOrder`
- `abs`, `min` and `max`
- The Art of Capturing Premises (TAOCP)
- Wheelchair tactics

## 1 Inequality

### 1.1 Basics

Inequality is determined by a partial order `PartialOrder`. A partial order is a relation with reflexivity, antisymmetry, and transitivity. In Lean, a relation means `α → α → Prop` for some type `α`, capturing the fact that each `a ≤ b` gives a proposition.

```
section

variable (a b c d : ℚ)
```

`PartialOrder` makes `LE`(`≤`) and `LT`(`<`) available in the context.

```
#check PartialOrder

#check a ≤ b
#check a < b
#check b ≥ a
#check b > a

#check le_refl
#check le_antisymm
#check le_trans

#check lt_irrefl
#check lt_asymm
#check lt_trans
```

`<` is determined by `≤`

```
#check lt_iff_le_not_ge
```

`≥`, `>` are just aliases of `≤`, `<`

```
example : (a < b) = (b > a) := by rfl
example : (a ≤ b) = (b ≥ a) := by rfl

example : a < b ↔ a ≤ b ∧ a ≠ b := by
  rw [lt_iff_le_not_ge]
  constructor
  · intro ⟨hab, hnba⟩
    constructor
    · exact hab
    · intro h
      rw [h] at hnba
      apply hnba
      exact le_refl b
  · intro ⟨hab, hnab⟩
    constructor
    · exact hab
    · intro hba
      apply hnab
      exact le_antisymm hab hba
#check lt_of_le_of_ne -- this have a related theorem
```

A linearly ordered commutative ring is a commutative ring with a total order s.t addition and multiplication are strictly monotone, e.g. $\mathbb{Q}$.

In Lean this reads `[CommRing R] [LinearOrder R] [IsStrictOrderedRing R]`.

We will work with $\mathbb{Q}$ as an example afterwards.

[TODO] For some reason, `LinearOrder ℚ` is constructed using classical logic. Don't be surprised if `#print axioms ...` shows some classical axioms.

## 1.2 Pure partial order reasoning

`norm_num` tactic solves numerical equalities and inequalities automatically.

```
#help tactic norm_num
example : (22 / 7 : ℚ) < 4 := by norm_num

-- [EXR]
example (hab : a ≤ b) (hba : b ≤ a) : a = b := by
  apply le_antisymm
  · exact hab
  · exact hba
```

`grw` rewrites like `rw`, but works for inequalities.

```
#help tactic grw
example (hab : a ≤ b) (hbc : b < c) : a < c := by
  grw [hab]
  exact hbc
example (hab : a ≤ b) (hbc : b < c) : a < c := by
  grw [← hab] at hbc
  exact hbc
#check lt_of_le_of_lt -- this have a name
```

`calc` is a term / tactic for proving inequalities by chaining.

```
#help tactic calc
example (hab : a ≤ b) (hbc : b < c) : a < c := by
  calc
    a ≤ b := hab
    _ < c := hbc
```

## 1.3 Linear order reasoning

A linear order is a partial order with `le_total`: either `a ≤ b` or `b ≤ a`.

```
#check le_total
```

[EXR] Use this to prove the trichotomy of `<` and `=`.

```
example : a < b ∨ a = b ∨ a > b := by
  rcases le_total a b with (hle | h)
  · by_cases heq : a = b
    · right; left; exact heq
    · left
      apply lt_of_le_of_ne
      · exact hle
      · exact heq
  · -- do it similarly
    sorry
#check eq_or_lt_of_le -- this have a name
```

## 1.4 Monotonicity of `+`

It's important to recognize that the (strict) monotonicity of `+` is a nontrivial theorem. That is a part of the meaning of `IsStrictOrderedRing`.

```
#synth IsStrictOrderedRing ℚ

#check add_le_add_left
#check add_le_add_right

#check add_lt_add_left
#check add_lt_add_right
```

Luckily, `grw` recognizes these theorems and applies them automatically.
transposition of `≤`

```
example (h : a + b ≤ c) : a ≤ c - b := by
  grw [← h]
  simp
```

monotonicity of `+`

```
example : a + c ≤ b + c ↔ a ≤ b := by
  constructor
  · intro h
    calc
```

```
      a = (a + c) - c := by simp
      _ ≤ (b + c) - c := by grw [h]
      _ = b := by simp
   · intro h
    grw [h]
#check add_le_add_iff_right -- this have a name
```

strict monotonicity of `+`

```
example : a < b ↔ a + c < b + c := by
  constructor
  · contrapose!
    intro h
    rw [add_le_add_iff_right] at h
    exact h
  · contrapose!
    intro h
    grw [h]
#check add_lt_add_iff_right -- this have a name

-- [EXR]
example (h : a + b < c + d) : a - d < c - b := by
  sorry
```

## 1.5 Automation

Tired of these? Use automation!

### 1.5.1 `linarith`

`linarith` is a powerful tactic that solves linear inequalities automatically. It uses hypotheses in the context and basic properties of linear orders to deduce the goal.

`linarith only [h1, h2, ...]` use only hypotheses `h1`, `h2`, … to solve the goal.

```
#help tactic linarith

example : a < b ↔ a - c < b - c := by
  constructor
  all_goals
    intro
    linarith

example (h : a + b < c + d) : a - d < c - b := by
  linarith

example (h : a > 0) : (2 / 3) * a > 0 := by
  linarith

example (h : (-5 / 3) * a > 0) : 4 * a < 0 := by
  linarith
```

Note the limitations of `linarith`.

It only works for linear inequalities, not polynomial ones.

```
example : a ^ 2 ≥ 0 := by sorry -- linarith fails here
```

though some of polynomial inequalities can be solved by `nlinarith`

```
#help tactic nlinarith
example : a ^ 2 ≥ 0 := by nlinarith
#check sq_nonneg -- this have a name
```

It solve all inequalities in a dense linear order.

It does solve some inequalities in discrete linear orders like $\mathbb{Z}$, but no guarantee for all of them.

```
example (n m : ℤ) (h : n < m) : n + 1 ≤ m := by linarith
example (n m : ℤ) (h : n < m) : n + (1/2 : ℚ) ≤ m := by sorry -- linarith fails here
```

It won't recognize inequalities involving `min`, `max`, `abs`, etc. It won't recognize some basic `simp` transformations, either.

```
example (h : a * (min 1 2) > 0) : (id a) ≥ 0 := by
  simp at *
  linarith -- direct `linarith` will fail
```

[EXR] admits a dense linear order

```
example (hab : a < b) : ∃ c : ℚ, a < c ∧ c < b := by
  use (a + b) / 2
  constructor
  all_goals linarith
```

### 1.5.2 `simp`

`add_lt_add_iff_right`-like theorems are registered for `simp`, so sometimes `simp` can reduce things like:

```
example (h : a + b < c + b) : a < c := by
  simp at h
  exact h
```

### 1.5.3 `apply_fun`

Sometimes you would like to `apply_fun` at an inequality. This requires you to manually show the monotonicity of the function.

```
example (h : a + b < c + d) : a - d < c - b := by
  apply_fun ( · - b - d) at h
  · ring_nf at *
    exact h
  · unfold StrictMono
    simp
```

[EXR] Mimick the above example.

```
example (h : a + c ≤ b) : a ≤ b - c := by
  apply_fun (· - c) at h
  · ring_nf at *
    exact h
  · unfold Monotone
    simp
```

### 1.6 Monotonicity of `*`

[TODO] It's not needed in the course so far, so we skip it for now.

```
end
```

## 2 `abs`, `min`, `max` and TAOCP

A mature formalizer finds their theorems by themselves. The art of capturing premises includes, but not limited to:

- `exact?`
- name guessing
- natural language search engine: LeanSearch, LeanExplore, etc.
- mathlib documentation
- AI copilot completion

```
section

variable (a b c : ℚ)

#check abs
```

[EXR] Find all the below by yourself

```
example : |a| ≥ 0 := by exact abs_nonneg a
example : |-a| = |a| := by exact abs_neg a
example : |a * b| = |a| * |b| := by exact abs_mul a b
example : |a + b| ≤ |a| + |b| := by exact abs_add_le a b
example : |a| - |b| ≤ |a - b| := by exact abs_sub_abs_le_abs_sub a b
example : |a| ≤ b ↔ -b ≤ a ∧ a ≤ b := by exact abs_le
example : |a| ≥ b ↔ a ≤ -b ∨ b ≤ a := by exact le_abs'

example (h : a ≥ 0) : |a| = a := by exact abs_of_nonneg h
example (h : a ≤ 0) : |a| = -a := by exact abs_of_nonpos h
example (h : b ≥ 0) : |a| = b ↔ a = b ∨ a = -b := by exact abs_eq h
```

A mindless way to prove these linear inequalities involving `abs` is to eliminate all `abs` by casing on the sign of the arguments, then use `linarith`.

```
example : |a - c| ≤ |a - b| + |b - c| := by
  all_goals rcases le_total 0 (a - b) with h1 | h1
  all_goals
    try rw [abs_of_nonneg h1]
```

```
    try rw [abs_of_nonpos h1]
  all_goals rcases le_total 0 (b - c) with h2 | h2
  all_goals
    try rw [abs_of_nonneg h2]
    try rw [abs_of_nonpos h2]
  all_goals rcases le_total 0 (a - c) with h3 | h3
  all_goals
    try rw [abs_of_nonneg h3]
    try rw [abs_of_nonpos h3]

  all_goals linarith
```

combine brute-force method with theorem-finding

```
example : |(|a| - |b|)| ≤ |a - b| := by
  rcases le_total 0 (|a| - |b|) with h1 | h1
  all_goals
    try rw [abs_of_nonneg h1]
    try rw [abs_of_nonpos h1]
   · apply abs_sub_abs_le_abs_sub
   · simp only [neg_sub] -- use `simp only` to supress unwanted lemmas
    grw [abs_sub_abs_le_abs_sub]
    rw [← abs_neg]
    simp

-- [EXR]
example : |(|a| - |b|)| ≤ |a + b| := by
  rcases le_total 0 (|a| - |b|) with h1 | h1
  all_goals
    try rw [abs_of_nonneg h1]
    try rw [abs_of_nonpos h1]
   · haveI := abs_sub_abs_le_abs_sub a (-b)
    simp at *
    exact this
   · haveI := abs_sub_abs_le_abs_sub b (-a)
    simp at *
    grw [this]
    ring_nf
    simp
```

[EXR] Get familiar with `min`, `max` and solve the following by yourself.

```
example : min a b ≤ a := by exact min_le_left a b
example : min a b + max a b = a + b := by exact min_add_max a b

end
```

## 3 Wheelchair tactics

You have seen some all-in-one tactics like `simp`, `ring` and `linarith`. There are even more powerful tactics that save your effort. Do try them when you feel tired of trivial steps.

```
#help tactic simp
#help tactic dsimp
#help tactic simp_rw
#help tactic field_simp

#help tactic group
#help tactic abel
#help tactic ring
#help tactic module

#help tactic linarith
#help tactic nlinarith

#help tactic omega
#help tactic aesop
#help tactic grind
#help tactic tauto
```

A tactic cheatsheet is available at lean-tactics.pdf