Matemáticas en Lean

José A. Alonso Jiménez

Grupo de Lógica Computacional
Dpto. de Ciencias de la Computación e Inteligencia Artificial
Universidad de Sevilla
Sevilla, 3 de agosto de 2020

Índice general

Capítulo 1

Introducción

1.1. Resumen

El objetivo de este trabajo es presentar el uso de Lean mediante ejemplos matemáticos usando. Está basado en el libro Mathematics in Lean de Jeremy Avigad, Kevin Buzzard, Robert Y. Lewis y Patrick Massot.

El trabajo se presenta en formas:

- Como un libro en PDF
- Como una página HTML.
- Como un proyecto en GitHub.

1.2. Presentación panorámica de Lean

- Ejemplo de evaluación

```
-- ---------------------------------------------------------------------
-- Ejercicio. Calcular el valor de 2+3.
-- ---------------------------------------------------------------------

#eval 2 + 3

-- Comentario: Al poner el cursor sobre eval se escribe el resultado en el
-- minibuffer.
```

- Ejemplo de comprobación con check

```
-- ---------------------------------------------------------------------
-- Ejercicio: Calcular el tipo de la expresión 2+3.
-- ---------------------------------------------------------------------

#check 2 + 3

-- Comentario: Al colocar el cursor sobre check escribe en el minibuffer
--    2 + 3 : ℕ
-- que indica que el valor de la expresión es un número natural.
```

- Ejemplo de definición de funciones

```
-- ---------------------------------------------------------------------
-- Ejercicio 1. Definir la función f que le suma 3 a cada número natural.
-- ---------------------------------------------------------------------

def f (x : ℕ) := x + 3

-- ---------------------------------------------------------------------
-- Ejercicio 2. Calcular el tipo de f.
-- ---------------------------------------------------------------------

#check f

-- Comentario: Al colocar el cursor sobre check se obtiene
--    f : ℕ → ℕ

-- ---------------------------------------------------------------------
-- Ejercicio 3. Calcular el valor de f(2).
-- ---------------------------------------------------------------------

#eval f 2

-- Comentario: Al colocar el cursor sobre eval se obtiene
--    5
```

- Ejemplo de proposiciones

```
-- ---------------------------------------------------------------------
-- Ejercicio 1. Definir la proposión ultimo_teorema_de_Fermat que
-- expresa el último teorema de Fermat.
-- ---------------------------------------------------------------------

```

```
def ultimo_teorema_de_Fermat :=
  ∀ x y z n : ℕ, n > 2 → x * y * z ≠ 0 → x^n + y^n ≠ z^n

-- ---------------------------------------------------------------------
-- Ejercicio 2. Calcular el tipo de ultimo_teorema_de_Fermat
-- ---------------------------------------------------------------------

#check ultimo_teorema_de_Fermat

-- Comentario: Al colocar el cursor sobre check se obtiene
--    ultimo_teorema_de_Fermat : Prop
```

- Ejemplo de teoremas

```
-- ---------------------------------------------------------------------
-- Ejercicio 1. Demostrar el teorema facil que afirma que 2 + 3 = 5.
-- ---------------------------------------------------------------------

theorem facil : 2 + 3 = 5 := rfl

-- Comentarios:
-- 1. Para activar la ventana de objetivos (*Lean Goal*) se escribe
--    C-c C-g
-- 2. Se desactiva volviendo a escribir C-c C-g
-- 3. La táctica rfl (ver https://bit.ly/2BYbiBH) comprueba que 2+3 y 5
--    son iguales por definición.

-- ---------------------------------------------------------------------
-- Ejercicio 2. Calcular el tipo de facil
-- ---------------------------------------------------------------------

#check facil

-- Comentario: Colocando el cursor sobre check se obtiene
--    facil : 2 + 3 = 5

-- ---------------------------------------------------------------------
-- Ejercicio 3. Enunciar el teorema dificil que afirma que se verifica
-- el último teorema de Fermat, omitiendo la demostración.
-- ---------------------------------------------------------------------

def ultimo_teorema_de_Fermat :=
  ∀ x y z n : ℕ, n > 2 → x * y * z ≠ 0 → x^n + y^n ≠ z^n
```

```
theorem dificil : ultimo_teorema_de_Fermat := sorry

-- Comentarios:
-- 1. La palabra sorry se usa para omitir la demostración.
-- 2. Se puede verificar la teoría pulsando
--       C-x ! l
--    Se obtiene
--       31   1 warning         declaration 'dificil' uses sorry

-- ---------------------------------------------------------------------
-- Ejercicio 3. Calcular el tipo de dificil.
-- ---------------------------------------------------------------------

#check dificil

-- Comentario: Colocando el cursor sobre check se obtiene
--    dificil : ultimo_teorema_de_Fermat
```

- Ejemplo de demostración

```
-- ---------------------------------------------------------------------
-- Ejercicio. Demostrar que los procutos de los números naturales por
-- números pares son pares.
-- ---------------------------------------------------------------------

import data.nat.parity tactic

open nat

-- 1ª demostración
example : ∀ m n, even n → even (m * n) :=
  assume m n ⟨k, (hk : n = 2 * k)⟩,
  have hmn : m * n = 2 * (m * k),
    by rw [hk, mul_left_comm],
  show ∃ l, m * n = 2 * l,
    from ⟨_, hmn⟩

-- 2ª demostración (mediante término)
example : ∀ m n, even n → even (m * n) :=
  λ m n ⟨k, hk⟩, ⟨m * k, by rw [hk, mul_left_comm]⟩

-- 3ª demostración (mediante tácticas)
example : ∀ m n, even n → even (m * n) :=
  begin
```

```
    rintros m n ⟨k, hk⟩,
    use m * k,
    rw [hk, mul_left_comm]
  end

-- 4ª demostración (mediante tácticas en una línea)
example : ∀ m n, even n → even (m * n) :=
  by rintros m n ⟨k, hk⟩; use m * k; rw [hk, mul_left_comm]

-- 4ª demostración (automática)
example : ∀ m n, even n → even (m * n) :=
  by intros; simp * with parity_simps

-- Comentarios:
-- 1. Al poner el curso en la línea 1 sobre data.nat.parity y pulsar M-.
--    se abre la teoría correspondiente.
-- 2. Se activa la ventana de objetivos (*Lean Goal*) pulsando C-c C-g
-- 3. Al mover el cursor sobre las pruebas se actualiza la ventana de
--    objetivos.
```

Capítulo 2

Aspectos básicos del razonamiento matemático en Lean

En este capítulo se presentan los aspectos básicos del razonamiento matemático en Lean:

- cálculos,
- aplicación de lemas y teoremas y
- razonamiento sobre estructuras genéricas.

2.1. Cálculos

- Asociativa conmutativa de los reales

```
-- ---------------------------------------------------------------------
-- Ejercicio. Demostrar que los números reales tienen la siguente
-- propiedad
--    (a * b) * c = b * (a * c)
-- ---------------------------------------------------------------------

import data.real.basic

-- 1ª demostración
-- ===============

example (a b c : ℝ) : (a * b) * c = b * (a * c) :=
```

```
begin
  rw mul_comm a b,
  rw mul_assoc b a c,
end

-- Comentarios:
-- 1. Al colocar el cursor sobre el nombre de un lema se ve su enunciado.
-- 2. Para completar el nombre de un lema basta escribir parte de su
--    nombre y completar con S-SPC (es decir, simultáneamente las teclas
--    de mayúscula y la de espacio).
-- 3. los lemas usados son
--    + mul_com   : (∀ a b : G),   a * b = b * a
--    + mul_assoc : (∀ a b c : G), (a * b) * c = a * (b * c)
-- 4. La táctica (rw e) (ver https://bit.ly/3eUxbjD) reescribe una
--    expresión usando la ecuación e.

-- El desarrollo de la prueba es:
--
--    inicio
--       a b c : ℝ
--       ⊢ (a * b) * c = b * (a * c)
--    rw mul_comm a b,
--       a b c : ℝ
--       ⊢ (a * b) * c = b * (a * c)
--    rw mul_assoc b a c
--       no goals

-- 2ª demostración
-- ===============

example
  (a b c : ℝ)
  : (a * b) * c = b * (a * c) :=
calc
  (a * b) * c = (b * a) * c : by rw mul_comm a b
          ... = b * (a * c) : by rw mul_assoc b a c

-- Comentario: El entorno calc (ver https://bit.ly/2NRxiAU) permite
-- escribir demostraciones ecuacionales.

-- 3ª demostración
-- ===============

example (a b c : ℝ) : (a * b) * c = b * (a * c) :=
by ring
```

```
-- Comentario: La táctica ring demuestra ecuaciones aplicando las
-- propiedades de anillos.
```

- Ejercicios sobre aritmética real

```
-- ---------------------------------------------------------------------
-- Ejercicio 1. Demostrar que los números reales tienen la siguente
-- propiedad
--    (c * b) * a = b * (a * c)
-- ---------------------------------------------------------------------

-- Importación de librería
-- =======================

import data.real.basic

-- 1ª demostración
-- ===============

example
  (a b c : ℝ)
  : (c * b) * a = b * (a * c) :=
begin
  rw mul_comm c b,
  rw mul_assoc,
  rw mul_comm c a,
end

-- Desarrollo de la prueba:
-- ------------------------

--    a b c : ℝ
--    ⊢ (c * b) * a = b * (a * c)
-- rw mul_comm c b,
--    a b c : ℝ
--    ⊢ (b * c) * a = b * (a * c)
-- rw mul_assoc,
--    a b c : ℝ
--    ⊢ b * (c * a) = b * (a * c)
-- rw mul_comm c a,
--   no goals

-- 2ª demostración
```

```
-- ===============

example
  (a b c : ℝ)
  : (c * b) * a = b * (a * c) :=
calc
  (c * b) * a = (b * c) * a : by rw mul_comm c b
          ... = b * (c * a) : by rw mul_assoc
          ... = b * (a * c) : by rw mul_comm c a

-- 3ª demostración
-- ===============

example
  (a b c : ℝ)
  : (c * b) * a = b * (a * c) :=
by ring

-- ---------------------------------------------------------------------
-- Ejercicio 2. Demostrar que los números reales tienen la siguente
-- propiedad
--    a * (b * c) = b * (a * c)
-- ---------------------------------------------------------------------

-- 1ª demostración
-- ===============

example
  (a b c : ℝ)
  : a * (b * c) = b * (a * c) :=
begin
  rw ←mul_assoc,
  rw mul_comm a b,
  rw mul_assoc,
end

-- Comentario. Con la táctica (rw ←e) se aplica reescritura sustituyendo
-- el término derecho de la igualdad e por el izquierdo.

-- Desarrollo de la prueba
-- -----------------------

--    a b c : ℝ
--    ⊢ a * (b * c) = b * (a * c)
-- rw ←mul_assoc,
```

```
--    a b c : ℝ
--    ⊢ (a * b) * c = b * (a * c)
-- rw mul_comm a b,
--    a b c : ℝ
--    ⊢ (b * a) * c = b * (a * c)
-- rw mul_assoc,
--    no goals

-- 2ª demostración
-- ===============

example
  (a b c : ℝ)
  : a * (b * c) = b * (a * c) :=
calc
  a * (b * c) = (a * b) * c : by rw ←mul_assoc
          ... = (b * a) * c : by rw mul_comm a b
          ... = b * (a * c) : by rw mul_assoc

-- 3ª demostración
-- ===============

example
  (a b c : ℝ)
  : a * (b * c) = b * (a * c) :=
by ring
```

- Ejemplo de rw con hipótesis

```
-- ---------------------------------------------------------------------
-- Ejercicio. Demostrar que si a, b, c, d, e y f son números reales
-- tales que
--    a * b = c * d
--    e = f
-- Entonces,
--    a * (b * e) = c * (d * f)
-- ---------------------------------------------------------------------

import data.real.basic

-- 1ª demostración
-- ===============

example
```

```
  (a b c d e f : ℝ)
  (h1 : a * b = c * d)
  (h2 : e = f)
  : a * (b * e) = c * (d * f) :=
begin
  rw h2,
  rw ←mul_assoc,
  rw h1,
  rw mul_assoc,
end

-- Comentario: La táctica (rw h2) reescribe el objetivo con la igualdad
-- de la nipótesis h2.

-- Desarrollo de la prueba
-- -----------------------

-- inicio
--    a b c d e f : ℝ,
--    h1 : a * b = c * d,
--    h2 : e = f
--    ⊢ a * (b * e) = c * (d * f)
-- rw h2,
--    S
--    ⊢ a * (b * f) = c * (d * f)
-- rw ←mul_assoc,
--    S
--    ⊢ (a * b) * f = c * (d * f)
-- rw h1,
--    S
--    ⊢ (c * d) * f = c * (d * f)
-- rw mul_assoc,
--    no goals
--
-- En el desarrollo anterior, S es el conjunto de las hipótesis; es
-- decir,
--    S = {a b c d e f : ℝ,
--         h1 : a * b = c * d,
--         h2 : e = f}

-- 2ª demostración
-- ===============

example
  (a b c d e f : ℝ)
```

```
  (h1 : a * b = c * d)
  (h2 : e = f)
  : a * (b * e) = c * (d * f) :=
calc
  a * (b * e) = a * (b * f) : by rw h2
          ... = (a * b) * f : by rw ←mul_assoc
          ... = (c * d) * f : by rw h1
          ... = c * (d * f) : by rw mul_assoc

-- 3ª demostración
-- ===============

example
  (a b c d e f : ℝ)
  (h1 : a * b = c * d)
  (h2 : e = f)
  : a * (b * e) = c * (d * f) :=
by finish

-- Comentario: La táctica finish (ver https://bit.ly/3ionJHE) simplifica
-- el objetivo usando las hipótesis.
```

- Ejercicio de rw con hipótesis

```
-- ---------------------------------------------------------------------
-- Ejercicio 1. Demostrar que si a, b, c, d, e y f son números reales
-- tales que
--    b * c = e * f
-- entonces
--    ((a * b) * c) * d = ((a * e) * f) * d
-- ---------------------------------------------------------------------

import data.real.basic

-- 1ª demostración
-- ===============

example
  (a b c d e f : ℝ)
  (h : b * c = e * f)
  : ((a * b) * c) * d = ((a * e) * f) * d :=
begin
  rw mul_assoc a,
  rw h,
```

```
  rw ←mul_assoc a,
end

-- El desarrollo de la prueba es
--
-- inicio
--    a b c d e f : ℝ,
--    h : b * c = e * f
--    ⊢ (a * (b * c)) * d = ((a * e) * f) * d
-- rw mul_assoc a,
--    S
--    ⊢ a * (b * c) * d = a * e * f * d
-- rw h,
--    S
--    ⊢ a * (e * f) * d = a * e * f * d
-- rw ←mul_assoc a,
--    no goals
--
-- En el desarrollo anterior, S es el conjunto de hipótesis; es decir,
--    S = {a b c d e f : ℝ,
--         h : b * c = e * f}

-- 2ª demostración
-- ===============

example
  (a b c d e f : ℝ)
  (h : b * c = e * f)
  : ((a * b) * c) * d = ((a * e) * f) * d :=
calc
  ((a * b) * c) * d = (a * (b * c)) * d : by rw mul_assoc a
                ... = (a * (e * f)) * d : by rw h
                ... = ((a * e) * f) * d : by rw ←mul_assoc a

-- 3ª demostración
-- ===============

example
  (a b c d e f : ℝ)
  (h : b * c = e * f)
  : ((a * b) * c) * d = ((a * e) * f) * d :=
by finish

-- ---------------------------------------------------------------------
-- Ejercicio 2. Demostrar que si a, b, c y d son números reales tales
```

```
-- que
--    c = b * a - d
--    d = a * b
-- entonces
--    c = 0
-- ----------------------------------------------------------------------

-- 1ª demostración
-- ===============

example
  (a b c d : ℝ)
  (h1 : c = b * a - d)
  (h2 : d = a * b)
  : c = 0 :=
begin
  rw h1,
  rw mul_comm,
  rw h2,
  rw sub_self,
end

-- Comentario: El último lema se puede encontrar escribiendo previamente
--    by library_search
-- y afirma que
--    ∀ (a : G), a - a = 0

-- Desarrollo de la prueba:
--
-- inicio
--    a b c d : ℝ,
--    h1 : c = b * a - d,
--    h2 : d = a * b
--    ⊢ c = 0
-- rw h1,
--    S
--    ⊢ b * a - d = 0
-- rw mul_comm,
--    S
--    ⊢ a * b - d = 0
-- rw h2,
--    S
--    ⊢ a * b - a * b = 0
-- rw sub_self,
--    no goals
```

```
--
-- En el desarrollo anterior, S es el conjunto de hipótesis; es decir,
--    S = {a b c d : ℝ,
--         h1 : c = b * a - d,
--         h2 : d = a * b}

-- 2ª demostración
-- ===============

example
  (a b c d : ℝ)
  (h1 : c = b * a - d)
  (h2 : d = a * b)
  : c = 0 :=
calc
  c   = b * a - d     : by rw h1
  ... = a * b - d     : by rw mul_comm
  ... = a * b - a * b : by rw h2
  ... = 0             : by rw sub_self

-- 3ª demostración
-- ===============

example
  (a b c d : ℝ)
  (h1 : c = b * a - d)
  (h2 : d = a * b)
  : c = 0 :=
by finish [mul_comm]
```

- Reescritura con varios lemas

```
-- ----------------------------------------------------------------------
-- Ejercicio. Demostrar que si a, b, c, d, e y f son números reales
-- tales que
--    a * b = c * d
--    e = f
-- entonces
--    a * (b * e) = c * (d * f)
-- ----------------------------------------------------------------------

import data.real.basic

-- 1ª demostración
```

```
-- ===============

example
  (a b c d e f : ℝ)
  (h1 : a * b = c * d)
  (h2 : e = f)
  : a * (b * e) = c * (d * f) :=
begin
  rw h2,
  rw ←mul_assoc,
  rw h1,
  rw mul_assoc,
end

-- Desarrollo de la prueba
--
--    a b c d e f : ℝ,
--    h1 : a * b = c * d,
--    h2 : e = f
--    ⊢ a * (b * e) = c * (d * f)
-- rw h2,
--    ⊢ a * (b * f) = c * (d * f)
-- rw ←mul_assoc,
--    ⊢ (a * b) * f = c * (d * f)
-- rw h1,
--    ⊢ (c * d) * f = c * (d * f)
-- rw mul_assoc,
--    no goals

-- 2ª demostración
-- ===============

example
  (a b c d e f : ℝ)
  (h1 : a * b = c * d)
  (h2 : e = f)
  : a * (b * e) = c * (d * f) :=
begin
  rw [h2, ←mul_assoc, h1, mul_assoc]
end

-- Comentario: Colocando el cursor en las comas se observa el progreso
-- en la ventana de objetivos.

-- 3ª demostración
```

```
-- ===============

example
  (a b c d e f : ℝ)
  (h1 : a * b = c * d)
  (h2 : e = f)
  : a * (b * e) = c * (d * f) :=
by rw [h2, ←mul_assoc, h1, mul_assoc]

-- 4ª demostración
-- ===============

example
  (a b c d e f : ℝ)
  (h1 : a * b = c * d)
  (h2 : e = f)
  : a * (b * e) = c * (d * f) :=
by finish

-- 5ª demostración
-- ===============

example
  (a b c d e f : ℝ)
  (h1 : a * b = c * d)
  (h2 : e = f)
  : a * (b * e) = c * (d * f) :=
calc
  a * (b * e) = a * (b * f) : by rw h2
          ... = (a * b) * f : by rw ←mul_assoc
          ... = (c * d) * f : by rw h1
          ... = c * (d * f) : by rw mul_assoc
```

- Declaración de variables en secciones

```
-- ---------------------------------------------------------------------
-- Ejercicio 1. Importar la librería básica de los números reales.
-- ---------------------------------------------------------------------

import data.real.basic

-- ---------------------------------------------------------------------
-- Ejercicio 2. Crear una sección.
-- ---------------------------------------------------------------------
```

```
section

-- ---------------------------------------------------------------------
-- Ejercicio 3. Declarar que a, b y c son variables sobre los números
-- reales.
-- ---------------------------------------------------------------------

variables a b c : ℝ

-- ---------------------------------------------------------------------
-- Ejercicio 4. Calcular el tipo de a.
-- ---------------------------------------------------------------------

#check a

-- Comentario: Al colocar el cursor sobre check se obtiene
--    a : ℝ

-- ---------------------------------------------------------------------
-- Ejercicio 5. Calcular el tipo de a + b.
-- ---------------------------------------------------------------------

#check a + b

-- Comentario: Al colocar el cursor sobre check se obtiene
--    a + b : ℝ

-- ---------------------------------------------------------------------
-- Ejercicio 6. Comprobar que a es un número real.
-- ---------------------------------------------------------------------

#check (a : ℝ)

-- Comentario: Al colocar el cursor sobre check se obtiene
--    a : ℝ

-- ---------------------------------------------------------------------
-- Ejercicio 7. Calcular el tipo de
--    mul_comm a b
-- ---------------------------------------------------------------------

#check mul_comm a b

-- Comentario: Al colocar el cursor sobre check se obtiene
```

```
--    mul_comm a b : a * b = b * a

-- ---------------------------------------------------------------------
-- Ejercicio 8. Comprobar que el tipo de
--    mul_comm a b
-- es
--    a * b = b * a)
-- ---------------------------------------------------------------------

#check (mul_comm a b : a * b = b * a)

-- Comentario: Al colocar el cursor sobre check se obtiene
--    mul_comm a b : a * b = b * a

-- ---------------------------------------------------------------------
-- Ejercicio 9. Calcular el tipo de
--    mul_assoc c a b
-- ---------------------------------------------------------------------

#check mul_assoc c a b

-- Comentario: Al colocar el cursor sobre check se obtiene
--    mul_assoc c a b : c * a * b = c * (a * b)

-- ---------------------------------------------------------------------
-- Ejercicio 10. Calcular el tipo de
--    mul_comm a
-- ---------------------------------------------------------------------

#check mul_comm a

-- Comentario: Al colocar el cursor sobre check se obtiene
--    mul_comm a : ∀ (b : ℝ), a * b = b * a

-- ---------------------------------------------------------------------
-- Ejercicio 11. Calcular el tipo de
--    mul_comm
-- ---------------------------------------------------------------------

#check mul_comm

-- Comentario: Al colocar el cursor sobre check se obtiene
--    mul_comm : ∀ (a b : ?M_1), a * b = b * a

-- ---------------------------------------------------------------------
```

```
-- Ejercicio 12. Calcular el tipo de
--    @mul_comm
-- ----------------------------------------------------------------------

#check @mul_comm

-- Comentario: Al colocar el cursor sobre check se obtiene
--    mul_comm : ∀ {G : Type u_1} [_inst_1 : comm_semigroup G] (a b : G),
--                 a * b = b * a

end
```

- Demostración con calc

```
-- ----------------------------------------------------------------------
-- Ejercicio. Demostrar que si a y b son números reales, entonces
--    (a + b) * (a + b) = a * a + 2 * (a * b) + b * b
-- ----------------------------------------------------------------------

import data.real.basic

variables a b : ℝ

-- 1ª demostración
-- ===============

example :
  (a + b) * (a + b) = a * a + 2 * (a * b) + b * b :=
begin
  rw mul_add,
  rw add_mul,
  rw add_mul,
  rw ← add_assoc,
  rw add_assoc (a * a),
  rw mul_comm b a,
  rw ← two_mul,
end

-- El desarrollo de la prueba es
--
--    a b : ℝ
--    ⊢ (a + b) * (a + b) = a * a + 2 * (a * b) + b * b
-- rw mul_add,
--    ⊢ (a + b) * a + (a + b) * b = a * a + 2 * (a * b) + b * b
```

```
-- rw add_mul,
--    ⊢ a * a + b * a + (a + b) * b = a * a + 2 * (a * b) + b * b
-- rw add_mul,
--    ⊢ a * a + b * a + (a * b + b * b) = a * a + 2 * (a * b) + b * b
-- rw ← add_assoc,
--    ⊢ a * a + b * a + a * b + b * b = a * a + 2 * (a * b) + b * b
-- rw add_assoc (a * a),
--    ⊢ a * a + (b * a + a * b) + b * b = a * a + 2 * (a * b) + b * b
-- rw mul_comm b a,
--    ⊢ a * a + (a * b + a * b) + b * b = a * a + 2 * (a * b) + b * b
-- rw ← two_mul,
--    no goals

-- 2ª demostración
-- ===============

example :
  (a + b) * (a + b) = a * a + 2 * (a * b) + b * b :=
calc
  (a + b) * (a + b)
      = (a + b) * a + (a + b) * b       : by rw mul_add
  ... = a * a + b * a + (a + b) * b     : by rw add_mul
  ... = a * a + b * a + (a * b + b * b) : by rw add_mul
  ... = a * a + b * a + a * b + b * b   : by rw ← add_assoc
  ... = a * a + (b * a + a * b) + b * b : by rw add_assoc (a * a)
  ... = a * a + (a * b + a * b) + b * b : by rw mul_comm b a
  ... = a * a + 2 * (a * b) + b * b     : by rw ← two_mul

-- 3ª demostración
-- ===============

example :
  (a + b) * (a + b) = a * a + 2 * (a * b) + b * b :=
begin
  rw [mul_add, add_mul, add_mul],
  rw [←add_assoc, add_assoc (a * a)],
  rw [mul_comm b a, ←two_mul],
end

-- El desarrollo de la prueba es
--
--    a b : ℝ
--    ⊢ a * a + (a * b + a * b) + b * b = a * a + 2 * (a * b) + b * b
-- rw [mul_add, add_mul, add_mul],
--    ⊢ a * a + b * a + (a * b + b * b) = a * a + 2 * (a * b) + b * b
```

```
-- rw [←add_assoc, add_assoc (a * a)],
--    ⊢ a * a + (b * a + a * b) + b * b = a * a + 2 * (a * b) + b * b
-- rw [mul_comm b a, ←two_mul]
--    no goals

-- 4ª demostración
-- ===============

example :
  (a + b) * (a + b) = a * a + 2 * (a * b) + b * b :=
calc
  (a + b) * (a + b)
      = a * a + b * a + (a * b + b * b) : by rw [mul_add, add_mul, add_mul]
  ... = a * a + (b * a + a * b) + b * b : by rw [←add_assoc, add_assoc (a * a)]
  ... = a * a + 2 * (a * b) + b * b     : by rw [mul_comm b a, ←two_mul]

-- 4ª demostración
-- ===============

example :
  (a + b) * (a + b) = a * a + 2 * (a * b) + b * b :=
by ring
```

- Ejercicio con calc

```
-- ---------------------------------------------------------------------
-- Ejercicio. Demostrar que si a, b, c y d son números reales, entonces
--    (a + b) * (c + d) = a * c + a * d + b * c + b * d
-- ---------------------------------------------------------------------

import data.real.basic

variables a b c d : ℝ

-- 1ª demostración
-- ===============

example
  : (a + b) * (c + d) = a * c + a * d + b * c + b * d :=
begin
   rw add_mul,
   rw mul_add,
   rw mul_add,
   rw ← add_assoc,
```

```
end

-- El desarrollo de la prueba es
--
--    a b c d : ℝ
--    ⊢ (a + b) * (c + d) = a * c + a * d + b * c + b * d
-- rw add_mul,
--    ⊢ a * (c + d) + b * (c + d) = a * c + a * d + b * c + b * d
-- rw mul_add,
--    ⊢ a * c + a * d + b * (c + d) = a * c + a * d + b * c + b * d
-- rw mul_add,
--    ⊢ a * c + a * d + (b * c + b * d) = a * c + a * d + b * c + b * d
-- rw ← add_assoc,
--    no goals

-- 2ª demostración
-- ===============

example
  : (a + b) * (c + d) = a * c + a * d + b * c + b * d :=
calc
  (a + b) * (c + d)
      = a * (c + d) + b * (c + d)       : by rw add_mul
  ... = a * c + a * d + b * (c + d)     : by rw mul_add
  ... = a * c + a * d + (b * c + b * d) : by rw mul_add
  ... = a * c + a * d + b * c + b * d   : by rw ←add_assoc

-- 3ª demostración
-- ===============

example : (a + b) * (c + d) = a * c + a * d + b * c + b * d :=
by rw [add_mul, mul_add, mul_add, ←add_assoc]

-- 4ª demostración
-- ===============

example : (a + b) * (c + d) = a * c + a * d + b * c + b * d :=
by ring
```

- Ejercicio: Suma por diferencia

```
-- ----------------------------------------------------------------------
-- Ejercicio. Demostrar que si a y b son números reales, entonces
--    (a + b) * (a - b) = a^2 - b^2
```

```
-- ----------------------------------------------------------------------

import data.real.basic

variables a b c d : ℝ

-- 1ª demostración (por reescritura usando el lema anterior)
-- ==========================================================

-- El lema anterior es
lemma aux : (a + b) * (c + d) = a * c + a * d + b * c + b * d :=
by ring

-- La demostración es
example : (a + b) * (a - b) = a^2 - b^2 :=
begin
  rw sub_eq_add_neg,
  rw aux,
  rw mul_neg_eq_neg_mul_symm,
  rw add_assoc (a * a),
  rw mul_comm b a,
  rw neg_add_self,
  rw add_zero,
  rw ← pow_two,
  rw mul_neg_eq_neg_mul_symm,
  rw ← pow_two,
  rw ← sub_eq_add_neg,
end

-- El desarrollo de la demostración es
--    ⊢ (a + b) * (a - b) = a ^ 2 - b ^ 2
-- rw sub_eq_add_neg,
--    ⊢ (a + b) * (a + -b) = a ^ 2 - b ^ 2
-- rw aux,
--    ⊢ a * a + a * -b + b * a + b * -b = a ^ 2 - b ^ 2
-- rw mul_neg_eq_neg_mul_symm,
--    ⊢ a * a + -(a * b) + b * a + b * -b = a ^ 2 - b ^ 2
-- rw add_assoc (a * a),
--    ⊢ a * a + (-(a * b) + b * a) + b * -b = a ^ 2 - b ^ 2
-- rw mul_comm b a,
--    ⊢ a * a + (-(a * b) + a * b) + b * -b = a ^ 2 - b ^ 2
-- rw neg_add_self,
--    ⊢ a * a + 0 + b * -b = a ^ 2 - b ^ 2
-- rw add_zero,
--    ⊢ a * a + b * -b = a ^ 2 - b ^ 2
```

```
-- rw ← pow_two,
--    ⊢ a ^ 2 + b * -b = a ^ 2 - b ^ 2
-- rw mul_neg_eq_neg_mul_symm,
--    ⊢ a ^ 2 + -(b * b) = a ^ 2 - b ^ 2
-- rw ← pow_two,
--    ⊢ a ^ 2 + -b ^ 2 = a ^ 2 - b ^ 2
-- rw ← sub_eq_add_neg,
--    no goals

-- 3ª demostración (por calc)
-- ==========================

example : (a + b) * (a - b) = a^2 - b^2 :=
calc
  (a + b) * (a - b)
      = a * (a - b) + b * (a - b)         : by rw add_mul
  ... = (a * a - a * b) + b * (a - b)     : by rw mul_sub
  ... = (a^2 - a * b) + b * (a - b)       : by rw ← pow_two
  ... = (a^2 - a * b) + (b * a - b * b)   : by rw mul_sub
  ... = (a^2 - a * b) + (b * a - b^2)     : by rw ← pow_two
  ... = (a^2 + -(a * b)) + (b * a - b^2)  : by exact rfl
  ... = a^2 + (-(a * b) + (b * a - b^2))  : by rw add_assoc
  ... = a^2 + (-(a * b) + (b * a + -b^2)) : by exact rfl
  ... = a^2 + ((-(a * b) + b * a) + -b^2) : by rw ← add_assoc
                                             (-(a * b)) (b * a) (-b^2)
  ... = a^2 + ((-(a * b) + a * b) + -b^2) : by rw mul_comm
  ... = a^2 + (0 + -b^2)                  : by rw neg_add_self (a * b)
  ... = (a^2 + 0) + -b^2                  : by rw ← add_assoc
  ... = a^2 + -b^2                        : by rw add_zero
  ... = a^2 - b^2                         : by exact rfl

-- Comentario. Se han usado los siguientes lemas:
-- + pow_two a : a ^ 2 = a * a
-- + mul_sub a b c : a * (b - c) = a * b - a * c
-- + add_mul a b c : (a + b) * c = a * c + b * c
-- + add_sub a b c : a + (b - c) = a + b - c
-- + sub_sub a b c : a - b - c = a - (b + c)
-- + add_zero a : a + 0 = a

-- 4ª demostración
-- ===============

example : (a + b) * (a - b) = a^2 - b^2 :=
```

```
by ring
```

- Reescritura en hipótesis y táctica exact

```
-- ---------------------------------------------------------------------
-- Ejercicio. Demostrar que si a, b, c y d son números reales tales que
--    c = d * a + b
--    b = a * d
-- entonces
--    c = 2 * a * d
-- ---------------------------------------------------------------------

import data.real.basic

variables a b c d : ℝ

-- 1ª demostración
-- ===============

example
  (h1 : c = d * a + b)
  (h2 : b = a * d)
  : c = 2 * a * d :=
begin
  rw h2 at h1,
  clear h2,
  rw mul_comm d a at h1,
  rw ← two_mul (a*d) at h1,
  rewrite ← mul_assoc 2 a d at h1,
  exact h1,
end

-- Comentarios
-- 1. La táctica (rw e at h) rescribe la parte izquierda de la
--    ecuación e por la derecha en la hipótesis h.
-- 2. La táctica (exact p) tiene éxito si el tipo de p se unifica con el
--    objetivo.
-- 3. La táctica (clear h) borra la hipótesis h.

-- El desarrollo de la prueba es
--
--    a b c d : ℝ,
--    h1 : c = d * a + b,
--    h2 : b = a * d
```

```
--    ⊢ c = 2 * a * d
-- rw h2 at h1,
--    a b c d : ℝ,
--    h2 : b = a * d,
--    h1 : c = d * a + a * d
--    ⊢ c = 2 * a * d
-- clear h2,
--    a b c d : ℝ,
--    h1 : c = d * a + a * d
--    ⊢ c = 2 * a * d
-- rw mul_comm d a at h1,
--    a b c d : ℝ,
--    h1 : c = a * d + a * d
--    ⊢ c = 2 * a * d
-- rw ← two_mul (a*d) at h1,
--    a b c d : ℝ,
--    h1 : c = 2 * (a * d)
--    ⊢ c = 2 * a * d
-- rewrite ← mul_assoc 2 a d at h1,
--    a b c d : ℝ,
--    h1 : c = 2 * a * d
--    ⊢ c = 2 * a * d
-- exact h1
--    no goals

-- 2ª demostración
-- ===============

example
  (h1 : c = d * a + b)
  (h2 : b = a * d)
  : c = 2 * a * d :=
calc
  c = d * a + b       : by rw h1
  ... = d * a + a * d : by rw h2
  ... = a * d + a * d : by rw mul_comm d a
  ... = 2 * (a * d)   : by rw ← two_mul (a * d)
  ... = 2 * a * d     : by rw mul_assoc

-- 3ª demostración
-- ===============

example
  (h1 : c = d * a + b)
  (h2 : b = a * d)
```

```
  : c = 2 * a * d :=
by rw [h1, h2, mul_comm d a, ← two_mul (a * d), mul_assoc]

-- 4ª demostración
-- ===============

example
  (h1 : c = d * a + b)
  (h2 : b = a * d)
  : c = 2 * a * d :=
begin
  rw h1,
  rw h2,
  ring,
end

-- El desarrollo de la prueba es
--
--    a b c d : ℝ,
--    h1 : c = d * a + b,
--    h2 : b = a * d
--    ⊢ c = 2 * a * d
-- rw h1,
--    ⊢ d * a + b = 2 * a * d
-- rw h2,
--    ⊢ d * a + a * d = 2 * a * d
-- ring,
--    no goals

-- 5ª demostración
-- ===============

example
  (h1 : c = d * a + b)
  (h2 : b = a * d)
  : c = 2 * a * d :=
begin
  rw [h1, h2],
  ring,
end

-- 6ª demostración
-- ===============

example
```

```
  (h1 : c = d * a + b)
  (h2 : b = a * d)
  : c = 2 * a * d :=
by rw [h1, h2]; ring

-- 7ª demostración
-- ===============

example
  (h1 : c = d * a + b)
  (h2 : b = a * d)
  : c = 2 * a * d :=
by finish * using [two_mul]
```

- Demostraciones con ring

```
-- ---------------------------------------------------------------------
-- Ejercicio. Sean a, b, c y dnúmeros reales. Demostrar, con la táctica
-- ring, que
--    (c * b) * a = b * (a * c)
--    (a + b) * (a + b) = a * a + 2 * (a * b) + b * b
--    (a + b) * (a - b) = a^2 - b^2
-- Además, si
--    c = d * a + b
--    b = a * d
-- entonces
--    c = 2 * a * d
-- ---------------------------------------------------------------------

import data.real.basic

variables a b c d : ℝ

example : (c * b) * a = b * (a * c) :=
by ring

example : (a + b) * (a + b) = a * a + 2 * (a * b) + b * b :=
by ring

example : (a + b) * (a - b) = a^2 - b^2 :=
by ring

example
  (h1 : c = d * a + b)
```

```
  (h2 : b = a * d)
  : c = 2 * a * d :=
begin
  rw [h1, h2],
  ring
end
```

2.2. Demostraciones en estructuras algebraicas

2.2.1. Demostraciones en anillos

- Axiomas de anillos

```
-- ---------------------------------------------------------------------
-- Ejercicio 1. Importar la librería de anillos.
-- ---------------------------------------------------------------------

import algebra.ring

-- ---------------------------------------------------------------------
-- Ejercicio 2. Declarar R como un tipo sobre los anillos.
-- ---------------------------------------------------------------------

variables (R : Type*) [ring R]

-- ---------------------------------------------------------------------
-- Ejercicio 3. Comprobar que R verifica los axiomas de los anillos.
-- ---------------------------------------------------------------------

#check (add_assoc : ∀ a b c : R, a + b + c = a + (b + c))
#check (add_comm : ∀ a b : R, a + b = b + a)
#check (zero_add : ∀ a : R, 0 + a = a)
#check (add_left_neg : ∀ a : R, -a + a = 0)
#check (mul_assoc : ∀ a b c : R, a * b * c = a * (b * c))
#check (mul_one : ∀ a : R, a * 1 = a)
#check (one_mul : ∀ a : R, 1 * a = a)
#check (mul_add : ∀ a b c : R, a * (b + c) = a * b + a * c)
#check (add_mul : ∀ a b c : R, (a + b) * c = a * c + b * c)
```

- Propiedades de anillos conmutativos

```
-- ---------------------------------------------------------------------
-- Ejercicio 1. Importar la librería de las tácticas.
-- ---------------------------------------------------------------------

import tactic

-- ---------------------------------------------------------------------
-- Ejercicio 2. Declarar R como una variable de tipo de los anillos
-- conmutativos.
-- ---------------------------------------------------------------------

variables (R : Type*) [comm_ring R]

-- ---------------------------------------------------------------------
-- Ejercicio 3. Declarar a, b, c y d como variables sobre R.
-- ---------------------------------------------------------------------

variables a b c d : R

-- ---------------------------------------------------------------------
-- Ejercicio 4. Demostrar que
--     (c * b) * a = b * (a * c)
-- ---------------------------------------------------------------------

example : (c * b) * a = b * (a * c) :=
by ring

-- ---------------------------------------------------------------------
-- Ejercicio 5. Demostrar que
--     (a + b) * (a + b) = a * a + 2 * (a * b) + b * b
-- ---------------------------------------------------------------------

example : (a + b) * (a + b) = a * a + 2 * (a * b) + b * b :=
by ring

-- ---------------------------------------------------------------------
-- Ejercicio 6. Demostrar que
--     (a + b) * (a - b) = a^2 - b^2
-- ---------------------------------------------------------------------

example : (a + b) * (a - b) = a^2 - b^2 :=
by ring

-- ---------------------------------------------------------------------
-- Ejercicio 7. Demostrar que si
```

```
--    c = d * a + b
--    b = a * d
-- entonces
--    c = 2 * a * d
-- ----------------------------------------------------------------------

example
  (h1 : c = d * a + b)
  (h2 : b = a * d)
  : c = 2 * a * d :=
begin
  rw [h1, h2],
  ring
end
```

- Propiedades básicas de anillos

```
-- ----------------------------------------------------------------------
-- Ejercicio 1. Importar la teoría de anillos.
-- ----------------------------------------------------------------------

import algebra.ring

-- ----------------------------------------------------------------------
-- Ejercicio 2. Crear el espacio de nombres my_ring (para evitar
-- conflictos con los nombres).
-- ----------------------------------------------------------------------

namespace my_ring

-- ----------------------------------------------------------------------
-- Ejercicio 2. Declarar R como una variable implícita sobre los anillos.
-- ----------------------------------------------------------------------

variables {R : Type*} [ring R]

-- ----------------------------------------------------------------------
-- Ejercicio 3. Declarar a como una variable sobre R.
-- ----------------------------------------------------------------------

variable (a : R)

-- ----------------------------------------------------------------------
-- Ejercicio 4. Demostrar que
```

```
--    a + 0 = a
-- ---------------------------------------------------------------------

-- 1ª demostración
-- ===============

example : a + 0 = a :=
begin
  rw add_comm,
  rw zero_add,
end

-- El desarrollo de la prueba es
--
--    R : Type u_1,
--    _inst_1 : ring R,
--    a : R
--    ⊢ a + 0 = a
-- rw add_comm,
--    ⊢ 0 + a = a
-- rw zero_add,
--    no goals

-- 2ª demostración
-- ===============

example : a + 0 = a :=
calc
  a + 0 = 0 + a : by rw add_comm
    ... = a     : by rw zero_add


-- 3ª demostración
-- ===============

theorem add_zero : a + 0 = a :=
by rw [add_comm, zero_add]

-- ---------------------------------------------------------------------
-- Ejercicio 5. Demostrar que
--    a + -a = 0
-- ---------------------------------------------------------------------

-- 1ª demostración
-- ===============
```

```
example : a + -a = 0 :=
begin
  rw add_comm,
  rw add_left_neg,
end

-- El desarrollo de la prueba es
--
--    R : Type u_1,
--    _inst_1 : ring R,
--    a : R
--    ⊢ a + -a = 0
-- rw add_comm,
--    ⊢ -a + a = 0
-- rw add_left_neg,
--    no goals

-- 2ª demostración
-- ===============

example : a + -a = 0 :=
calc
  a + -a = -a + a : by rw add_comm
     ... = 0 : by rw add_left_neg

-- 3ª demostración
-- ===============

theorem add_right_neg : a + -a = 0 :=
by rw [add_comm, add_left_neg]


-- ---------------------------------------------------------------------
-- Ejercicio 6. Cerrar el espacio de nombre my_ring.
-- ---------------------------------------------------------------------

end my_ring

-- ---------------------------------------------------------------------
-- Ejercicio 7. Calcular el tipo de @my_ring.add_zero.
-- ---------------------------------------------------------------------

#check @my_ring.add_zero
```

```
-- Comentario: Al colocar el cursor sobre check se obtiene
--    my_ring.add_zero : ∀ {R : Type u_1} [_inst_1 : ring R] (a : R),
--                       a + 0 = a

-- ---------------------------------------------------------------------
-- Ejercicio 8. Calcular el tipo de @add_zero.
-- ---------------------------------------------------------------------

#check @add_zero

-- Comentario: Al colocar el cursor sobre check se obtiene
--    add_zero : ∀ {M : Type u_1} [_inst_1 : add_monoid M] (a : M),
--               a + 0 = a
```

- Lema neg_add_cancel_left

```
-- ---------------------------------------------------------------------
-- Ejercicio 1. Importar la teoría de anillos.
-- ---------------------------------------------------------------------

import algebra.ring

-- ---------------------------------------------------------------------
-- Ejercicio 2. Crear el espacio de nombre my_ring
-- ---------------------------------------------------------------------

namespace my_ring

-- ---------------------------------------------------------------------
-- Ejercicio 3. Declarar R como una variable sobre anillos.
-- ---------------------------------------------------------------------

variables {R : Type*} [ring R]

-- ---------------------------------------------------------------------
-- Ejercicio 5. Demostrar que para todo a, b ∈ R,
--    -a + (a + b) = b
-- ---------------------------------------------------------------------

-- 1ª demostración
-- ===============

example
  (a b : R)
```

```
  : -a + (a + b) = b :=
calc
  -a + (a + b) = (-a + a) + b : by rw ← add_assoc
           ... = 0 + b        : by rw add_left_neg
           ... = b            : by rw zero_add

-- 2ª demostración
-- ===============

theorem neg_add_cancel_left
  (a b : R)
  : -a + (a + b) = b :=
by rw [←add_assoc, add_left_neg, zero_add]

-- El desarrollo de la prueba es
--
--    R : Type u_1,
--    _inst_1 : ring R,
--    a b : R
--    ⊢ -a + (a + b) = b
-- rw ← add_assoc,
--    ⊢ -a + a + b = b
-- rw add_left_neg,
--    ⊢ 0 + b = b
-- rw zero_add,
--    no goals

-- ---------------------------------------------------------------------
-- Ejercicio 6. Cerrar el espacio de nombre my_ring.
-- ---------------------------------------------------------------------

end my_ring
```

- Ejercicio neg_add_cancel_right

```
-- ---------------------------------------------------------------------
-- Ejercicio 1. Importar la teoría de anillos.
-- ---------------------------------------------------------------------

import algebra.ring

-- ---------------------------------------------------------------------
-- Ejercicio 2. Crear el espacio de nombre my_ring.
-- ---------------------------------------------------------------------
```

```
namespace my_ring

-- ---------------------------------------------------------------------
-- Ejercicio 3. Declara R una variable sobre anillos.
-- ---------------------------------------------------------------------

variables {R : Type*} [ring R]

-- ---------------------------------------------------------------------
-- Ejercicio 4. Declarar a y b como variables sobre R.
-- ---------------------------------------------------------------------

variables a b : R

-- ---------------------------------------------------------------------
-- Ejercicio 5. Demostrar que
--     (a + b) + -b = a
-- ---------------------------------------------------------------------

-- 1ª demostración
-- ===============

example : (a + b) + -b = a :=
begin
  rw add_assoc,
  rw add_right_neg,
  rw add_zero,
end

-- El desarrollo de la prueba es
--
--    R : Type u_1,
--    _inst_1 : ring R,
--    a b : R
--    ⊢ a + b + -b = a
-- rw add_assoc,
--    ⊢ a + (b + -b) = a
-- rw add_right_neg,
--    ⊢ a + 0 = a
-- rw add_zero,
--    no goals

-- 2ª demostración
-- ===============
```

```
example : (a + b) + -b = a :=
by rw [add_assoc, add_right_neg, add_zero]

-- 3ª demostración
-- ===============

theorem neg_add_cancel_right : (a + b) + -b = a :=
calc
  (a + b) + -b
      = a + (b + -b) : by rw add_assoc
  ... = a + 0        : by rw add_right_neg
  ... = a            : by rw add_zero

-- ---------------------------------------------------------------------
-- Ejercicio 4. Cerrar la teoría my_ring
-- ---------------------------------------------------------------------

end my_ring
```

- Ejercicio: Cancelativas de la suma

```
-- ---------------------------------------------------------------------
-- Ejercicio 1. Importar la teoría de anillos.
-- ---------------------------------------------------------------------

import algebra.ring
import tactic

-- ---------------------------------------------------------------------
-- Ejercicio 2. Crear el espacio de nombre my_ring.
-- ---------------------------------------------------------------------

namespace my_ring

-- ---------------------------------------------------------------------
-- Ejercicio 3. Declara R una variable sobre anillos.
-- ---------------------------------------------------------------------

variables {R : Type*} [ring R]

-- ---------------------------------------------------------------------
-- Ejercicio 4. Declarar a, b y c como variables sobre R.
-- ---------------------------------------------------------------------
```

```
variables {a b c : R}

-- ---------------------------------------------------------------------
-- Ejercicio 5. Demostrar que si
--    a + b = a + c
-- entonces
--    b = c
-- ---------------------------------------------------------------------

-- 1ª demostración
-- ===============

theorem add_left_cancel
  (h : a + b = a + c)
  : b = c :=
calc
  b
      = 0 + b        : by rw zero_add
  ... = (-a + a) + b : by rw add_left_neg
  ... = -a + (a + b) : by rw add_assoc
  ... = -a + (a + c) : by rw h
  ... = (-a + a) + c : by rw ←add_assoc
  ... = 0 + c        : by rw add_left_neg
  ... = c            : by rw zero_add

-- 2ª demostración
-- ===============

example
  (h : a + b = a + c)
  : b = c :=
begin
  have h1 : -a + (a + b) = -a + (a + c),
    { by exact congr_arg (has_add.add (-a)) h, },
  clear h,
  rw ← add_assoc at h1,
  rw add_left_neg at h1,
  rw zero_add at h1,
  rw ← add_assoc at h1,
  rw add_left_neg at h1,
  rw zero_add at h1,
  exact h1,
end
```

```
-- El desarrollo de la prueba es
--
--    R : Type u_1,
--    _inst_1 : ring R,
--    a b c : R,
--    h : a + b = a + c
--    ⊢ b = c
-- have h1 : -a + (a + b) = -a + (a + c),
--    |    ⊢ -a + (a + b) = -a + (a + c)
--    | { by exact congr_arg (has_add.add (-a)) h },
--    h : a + b = a + c,
--    h1 : -a + (a + b) = -a + (a + c)
--    ⊢ b = c
-- clear h,
--    h1 : -a + (a + b) = -a + (a + c)
--    ⊢ b = c
-- rw ← add_assoc at h1,
--    h1 : -a + a + b = -a + (a + c)
--    ⊢ b = c
-- rw add_left_neg at h1,
--    h1 : 0 + b = -a + (a + c)
--    ⊢ b = c
-- rw zero_add at h1,
--    h1 : b = -a + (a + c)
--    ⊢ b = c
-- rw ← add_assoc at h1,
--    h1 : b = -a + a + c
--    ⊢ b = c
-- rw add_left_neg at h1,
--    h1 : b = 0 + c
--    ⊢ b = c
-- rw zero_add at h1,
--    h1 : b = c
--    ⊢ b = c
-- exact h1,
--    no goals

-- 3ª demostración
-- ===============

example
  (h : a + b = a + c)
  : b = c :=
by finish
```

```
-- ---------------------------------------------------------------------
-- Ejercicio 6. Demostrar que si
--    a + b = c + b
-- entonces
--    a = c
-- ---------------------------------------------------------------------

-- 1ª demostración
-- ===============

theorem add_right_cancel
  (h : a + b = c + b)
  : a = c :=
calc
  a
      = a + 0         : by rw add_zero
  ... = a + (b + -b)  : by rw add_right_neg
  ... = (a + b) + -b  : by rw add_assoc
  ... = (c + b) + -b  : by rw h
  ... = c + (b + -b)  : by rw ← add_assoc
  ... = c + 0         : by rw ← add_right_neg
  ... = c             : by rw add_zero

-- 2ª demostración
-- ===============

example
  (h : a + b = c + b)
  : a = c :=
by finish

-- ---------------------------------------------------------------------
-- Ejercicio 7. Cerrar el espacio de nombre my_ring.
-- ---------------------------------------------------------------------

end my_ring
```

- Lema mul_zero con have

```
-- ---------------------------------------------------------------------
-- Ejercicio. Demostrar que en los anillos
--    a * 0 = 0
-- ---------------------------------------------------------------------
```

```
import algebra.ring

namespace my_ring

variables {R : Type*} [ring R]

variable (a : R)

-- 1ª demostración
-- ===============

theorem mul_zero : a * 0 = 0 :=
begin
  have h : a * 0 + a * 0 = a * 0 + 0,
  { rw [←mul_add, add_zero, add_zero] },
  rw add_left_cancel h
end

-- 2ª demostración
-- ===============

example : a * 0 = 0 :=
begin
  have h : a * 0 + a * 0 = a * 0 + 0,
    calc a * 0 + a * 0 = a * (0 + 0) : by rw ←mul_add
                   ... = a * 0       : by rw add_zero
                   ... = a * 0 + 0   : by rw add_zero,
  rw add_left_cancel h
end

end my_ring
```

- Ejercicio zero_mul

```
-- ---------------------------------------------------------------------
-- Ejercicio. Demostrar que en los anillos,
--    0 * a = 0
-- ---------------------------------------------------------------------

import algebra.ring

namespace my_ring

variables {R : Type*} [ring R]
```

```
variable (a : R)

theorem zero_mul : 0 * a = 0 :=
have 0 * a + 0 = 0 * a + 0 * a, from calc
  0 * a + 0 = (0 + 0) * a   : by simp
        ... = 0 * a + 0 * a : by rw add_mul,
show 0 * a = 0, from (add_left_cancel this).symm

end my_ring
```

- Ejercicios sobre anillos

```
-- ---------------------------------------------------------------------
-- Ejercicio 1. Realizar las siguientes acciones:
--    1. Importar la teoría de anillos.
--    2. Crear el espacio de nombres my_ring
--    3. Declarar R como una variable sobre anillos.
--    4. Declarar a y b como variables sobre R.
-- ---------------------------------------------------------------------

import algebra.ring            -- 1
namespace my_ring              -- 2
variables {R : Type*} [ring R] -- 3
variables {a b : R}            -- 4

-- ---------------------------------------------------------------------
-- Ejercicio 2. Demostrar que si
--    a + b = 0
-- entonces
--    -a = b
-- ---------------------------------------------------------------------

theorem neg_eq_of_add_eq_zero
  (h : a + b = 0)
  : -a = b :=
calc
  -a  = -a + 0       : by rw add_zero
  ... = -a + (a + b) : by rw h
  ... = b            : by rw neg_add_cancel_left

-- ---------------------------------------------------------------------
-- Ejercicio 3. Demostrar que
--     (a + b) + -b = a
```

```
-- ----------------------------------------------------------------------

-- 1ª demostración
example : (a + b) + -b = a :=
calc (a + b) + -b = a + (b + -b) : by rw add_assoc
              ... = a + 0        : by rw add_right_neg
              ... = a            : by rw add_zero

-- 2ª demostración
lemma neg_add_cancel_right : (a + b) + -b = a :=
by rw [add_assoc, add_right_neg, add_zero]

-- ----------------------------------------------------------------------
-- Ejercicio 4. Demostrar que si
--    a + b = 0
-- entonces
--    a = -b
-- ----------------------------------------------------------------------

theorem eq_neg_of_add_eq_zero
  (h : a + b = 0)
  : a = -b :=
calc
  a   = (a + b) + -b : by rw neg_add_cancel_right
  ... = 0 + -b       : by rw h
  ... = -b           : by rw zero_add

-- ----------------------------------------------------------------------
-- Ejercicio 5. Demostrar que
--    - 0 = 0
-- ----------------------------------------------------------------------

theorem neg_zero : (-0 : R) = 0 :=
begin
  apply neg_eq_of_add_eq_zero,
  rw add_zero,
end

-- El desarrollo de la prueba es
--
--    R : Type u_1,
--    _inst_1 : ring R
--    ⊢ -0 = 0
-- apply neg_eq_of_add_eq_zero,
--    ⊢ 0 + 0 = 0
```

```
-- rw add_zero,
--    no goals

-- ----------------------------------------------------------------------
-- Ejercicio 6. Demostrar que
--     -(-a) = a
-- ----------------------------------------------------------------------

theorem neg_neg : -(-a) = a :=
begin
  apply neg_eq_of_add_eq_zero,
  rw add_left_neg,
end

-- El desarrollo de la prueba es
--
--    R : Type u_1,
--    _inst_1 : ring R,
--    a : R
--    ⊢ - -a = a
-- apply neg_eq_of_add_eq_zero,
--    ⊢ -a + a = 0
-- rw add_left_neg,
--    no goals

end my_ring
```

- Subtracción en anillos

```
-- ----------------------------------------------------------------------
-- Ejercicio 1. Realizar las siguientes acciones:
--    1. Importar la teoría de anillos.
--    2. Crear el espacio de nombres my_ring
--    3. Declarar R como una variable sobre anillos.
--    4. Declarar a y b como variables sobre R.
-- ----------------------------------------------------------------------

import algebra.ring            -- 1
namespace my_ring              -- 2
variables {R : Type*} [ring R] -- 3
variables (a b : R)            -- 4

-- ----------------------------------------------------------------------
-- Ejercicio 2. Demostrar que
```

```
--    a - b = a + -b
-- ---------------------------------------------------------------------

-- 1ª demostración
theorem sub_eq_add_neg  : a - b = a + -b :=
rfl

-- 2ª demostración
example : a - b = a + -b :=
by reflexivity

end my_ring
```

- Ejercicio self_sub

```
-- ---------------------------------------------------------------------
-- Ejercicio 1. Realizar las siguientes acciones:
--    1. Importar la teoría de anillos.
--    2. Crear el espacio de nombres my_ring
--    3. Declarar R como una variable sobre anillos.
--    4. Declarar a como variable sobre R.
-- ---------------------------------------------------------------------

import algebra.ring            -- 1
namespace my_ring              -- 2
variables {R : Type*} [ring R] -- 3
variables (a : R)              -- 4

-- ---------------------------------------------------------------------
-- Ejercicio 2. Demostrar que
--     a - a = 0
-- ---------------------------------------------------------------------


theorem self_sub : a - a = 0 :=
calc
  a - a = a + -a : by reflexivity
  ...   = 0      : by rw add_right_neg

end my_ring
```

- Ejercicio two_mul

```
-- ----------------------------------------------------------------------
-- Ejercicio 1. Realizar las siguientes acciones:
--    1. Importar la teoría de anillos.
--    2. Crear el espacio de nombres my_ring
--    3. Declarar R como una variable sobre anillos.
--    4. Declarar a como variable sobre R.
-- ----------------------------------------------------------------------

import algebra.ring            -- 1
namespace my_ring              -- 2
variables {R : Type*} [ring R] -- 3
variables (a : R)              -- 4

-- ----------------------------------------------------------------------
-- Ejercicio 2. Demostrar que
--    1 + 1 = 2
-- ----------------------------------------------------------------------

lemma one_add_one_eq_two : 1 + 1 = (2 : R) :=
by refl

-- ----------------------------------------------------------------------
-- Ejercicio 3. Demostrar que
--    2 * a = a + a
-- ----------------------------------------------------------------------

theorem two_mul : 2 * a = a + a :=
calc
  2 * a = (1 + 1) * a   : by rw one_add_one_eq_two
  ...   = 1 * a + 1 * a : by rw add_mul
  ...   = a + a         : by rw one_mul

end my_ring
```

2.2.2. Demostraciones en grupos

- Axiomas de grupo (versión aditiva)

```
-- ----------------------------------------------------------------------
-- Ejercicio 1. Importar la librería de grupos
-- ----------------------------------------------------------------------

import algebra.group
```

```
-- ---------------------------------------------------------------------
-- Ejercicio 2. Declarar A como un tipo sobre grupos aditivos.
-- ---------------------------------------------------------------------

variables (A : Type*) [add_group A]

-- ---------------------------------------------------------------------
-- Ejercicio 3. Comprobar que A verifica los axiomas de los grupos
-- ---------------------------------------------------------------------

#check (add_assoc :    ∀ a b c : A, a + b + c = a + (b + c))
#check (zero_add :     ∀ a : A,     0 + a = a)
#check (add_left_neg : ∀ a : A,     -a + a = 0)
```

- Axiomas de grupo multiplicativo

```
-- ---------------------------------------------------------------------
-- Ejercicio 1. Importar la librería de grupos
-- ---------------------------------------------------------------------

import algebra.group

-- ---------------------------------------------------------------------
-- Ejercicio 2. Declarar G como un tipo sobre grupos.
-- ---------------------------------------------------------------------

variables (G : Type*) [group G]

-- ---------------------------------------------------------------------
-- Ejercicio 3. Comprobar que G verifica los axiomas de los grupos
-- ---------------------------------------------------------------------

#check (mul_assoc :    ∀ a b c : G, a * b * c = a * (b * c))
#check (one_mul :      ∀ a : G,     1 * a = a)
#check (mul_left_inv : ∀ a : G,     a⁻¹ * a = 1)
```

- Ejercicios sobre grupos

```
-- ---------------------------------------------------------------------
-- Ejercicio 1. Realizar las siguientes acciones:
--    1. Importar la teoría de grupo.
--    2. Crear el espacio de nombres my_group
```

```
--    3. Declarar G como una variable sobre anillos.
--    4. Declarar a y b como variable sobre G.
-- ---------------------------------------------------------------------

import algebra.group            -- 1
variables {G : Type*} [group G] -- 2
namespace my_group              -- 3
variables (a b : G)             -- 4

-- ---------------------------------------------------------------------
-- Ejercicio 2. Demostrar que
--    a * a⁻¹ = 1
-- ---------------------------------------------------------------------

theorem mul_right_inv : a * a⁻¹ = 1 :=
calc
  a * a⁻¹ = 1 * (a * a⁻¹)                 : by rw one_mul
      ... = (1 * a) * a⁻¹                 : by rw mul_assoc
      ... = (((a⁻¹)⁻¹ * a⁻¹)  * a) * a⁻¹ : by rw mul_left_inv
      ... = ((a⁻¹)⁻¹ * (a⁻¹  * a)) * a⁻¹ : by rw ← mul_assoc
      ... = ((a⁻¹)⁻¹ * 1) * a⁻¹           : by rw mul_left_inv
      ... = (a⁻¹)⁻¹ * (1 * a⁻¹)           : by rw mul_assoc
      ... = (a⁻¹)⁻¹ * a⁻¹                 : by rw one_mul
      ... = 1                             : by rw mul_left_inv

-- ---------------------------------------------------------------------
-- Ejercicio 3. Demostrar que
--    a * 1 = a
-- ---------------------------------------------------------------------

theorem mul_one : a * 1 = a :=
calc
  a * 1 = a * (a⁻¹ * a) : by rw mul_left_inv
    ... = (a * a⁻¹) * a : by rw mul_assoc
    ... = 1 * a         : by rw mul_right_inv
    ... = a             : by rw one_mul

-- ---------------------------------------------------------------------
-- Ejercicio 4. Demostrar que si
--     b * a = 1
-- entonces
--    a⁻¹ = b
-- ---------------------------------------------------------------------
```

```
lemma inv_eq_of_mul_eq_one
  (h : b * a = 1)
  : a⁻¹ = b :=
calc
  a⁻¹ =  1 * a⁻¹       : by rw one_mul
  ... =  (b * a) * a⁻¹ : by rw h
  ... =  b * (a * a⁻¹) : by rw mul_assoc
  ... =  b * 1         : by rw mul_right_inv
  ... =  b             : by rw mul_one

-- ---------------------------------------------------------------------
-- Ejercicio 5. Demostrar que
--    (a * b)⁻¹ = b⁻¹ * a⁻¹
-- ---------------------------------------------------------------------

-- En la demostración se usará el siguiente lema:
lemma mul_inv_rev_aux : (b⁻¹ * a⁻¹) * (a * b) = 1 :=
calc
  (b⁻¹ * a⁻¹) * (a * b)
      = b⁻¹ * (a⁻¹ * (a * b)) : by rw mul_assoc
  ... = b⁻¹ * ((a⁻¹ * a) * b) : by rw mul_assoc
  ... = b⁻¹ * (1 * b)         : by rw mul_left_inv
  ... = b⁻¹ * b               : by rw one_mul
  ... = 1                     : by rw mul_left_inv

theorem mul_inv_rev : (a * b)⁻¹ = b⁻¹ * a⁻¹ :=
begin
  apply inv_eq_of_mul_eq_one,
  rw mul_inv_rev_aux,
end

-- El desarrollo de la prueba es
--
--    G : Type u_1,
--    _inst_1 : group G,
--    a b : G
--    ⊢ (a * b)⁻¹ = b⁻¹ * a⁻¹
-- apply inv_eq_of_mul_eq_one,
--    ⊢ b⁻¹ * a⁻¹ * (a * b) = 1
-- rw mul_inv_rev_aux,
--    no goals

-- ---------------------------------------------------------------------
-- Ejercicio 6.  Cerrar el espacio de nombre my_group.
-- ---------------------------------------------------------------------
```

```
end my_group
```

2.3. Uso de lemas y teoremas

- Propiedades reflexiva y transitiva

```
-- ---------------------------------------------------------------------
-- Ejercicio 1. Importar la teoría de los números reales.
-- ---------------------------------------------------------------------

import data.real.basic

-- ---------------------------------------------------------------------
-- Ejercicio 2. Declarar a, b y c como variables sobre los reales.
-- ---------------------------------------------------------------------

variables a b c : ℝ

-- ---------------------------------------------------------------------
-- Ejercicio 3. Declarar que
-- + h  es una variable de tipo a ≤ b
-- + h' es una variable de tipo b ≤ c
-- ---------------------------------------------------------------------

variables (h : a ≤ b) (h' : b ≤ c)

-- ---------------------------------------------------------------------
-- Ejercicio 4. Calcular el tipo de las siguientes expresiones:
--    + le_refl
--    + le_refl a
--    + le_trans
--    + le_trans h
--    + le_trans h h'
-- ---------------------------------------------------------------------

#check (le_refl : ∀ a : real, a ≤ a)
#check (le_refl a : a ≤ a)
#check (le_trans : a ≤ b → b ≤ c → a ≤ c)
#check (le_trans h : b ≤ c → a ≤ c)
#check (le_trans h h' : a ≤ c)
```

- Las tácticas apply y exact

```
-- ----------------------------------------------------------------------
-- Ejercicio 1. Realizar las siguientes acciones:
--    1. Importar la teoría de los números reales
--    2. Declarar x, y y z como variables sobre R.
-- ----------------------------------------------------------------------

import data.real.basic

variables (x y z : ℝ)

-- ----------------------------------------------------------------------
-- Ejercicio 2. Demostrar que si
--    x ≤ y
--    y ≤ z
-- entonces
--    x ≤ z
-- ----------------------------------------------------------------------

-- 1ª demostración
-- ===============

example
  (h1 : x ≤ y)
  (h2 : y ≤ z)
  : x ≤ z :=
begin
  apply le_trans,
  { apply h1, },
  apply h2,
end

-- El desarrollo de la prueba es
--
--    x y z : ℝ,
--    h1 : x ≤ y,
--    h2 : y ≤ z
--    ⊢ x ≤ z
-- apply le_trans,
--    ⊢ x ≤ ?m_1
-- { apply h1 },
--    ⊢ y ≤ z
-- apply h2,
--    no goals

-- 2ª demostración
```

```
-- ===============

example
  (h1 : x ≤ y)
  (h2 : y ≤ z)
  : x ≤ z :=
begin
  apply le_trans h1,
  apply h2,
end

-- El desarrollo de la prueba es
--
--    x y z : ℝ,
--    h1 : x ≤ y,
--    h2 : y ≤ z
--    ⊢ x ≤ z
-- apply le_trans h1,
--    ⊢ y ≤ z
-- apply h2,
--    no goals

-- 3ª demostración
-- ===============

example
  (h1 : x ≤ y)
  (h2 : y ≤ z)
  : x ≤ z :=
by exact le_trans h1 h2

-- 4ª demostración
-- ===============

example
  (h1 : x ≤ y)
  (h2 : y ≤ z)
  : x ≤ z :=
le_trans h1 h2

-- ---------------------------------------------------------------------
-- Ejercicio 4. Demostrar que
--    x ≤ x
-- ---------------------------------------------------------------------
```

```
-- 1ª demostración
example : x ≤ x :=
by apply le_refl

-- 2ª demostración
example : x ≤ x :=
by exact le_refl x

-- 3ª demostración
example (x : ℝ) : x ≤ x :=
le_refl x
```

- Propiedades del orden

```
-- ---------------------------------------------------------------------
-- Ejercicio 1. Realizar las siguientes acciones:
--    1. Importar la teoría de los números reales
--    2. Declarar a, b y c como variables sobre R.
-- ---------------------------------------------------------------------

import data.real.basic

variables a b c : ℝ

-- ---------------------------------------------------------------------
-- Ejercicio 2. Calcular el tipo de las siguientes expresiones:
--    + le_refl
--    + le_trans
--    + lt_of_le_of_lt
--    + lt_of_lt_of_le
--    + lt_trans
-- ---------------------------------------------------------------------


#check (le_refl  : ∀ a, a ≤ a)
#check (le_trans : a ≤ b → b ≤ c → a ≤ c)
#check (lt_of_le_of_lt : a ≤ b → b < c → a < c)
#check (lt_of_lt_of_le : a < b → b ≤ c → a < c)
#check (lt_trans : a < b → b < c → a < c)
```

- Ejercicio sobre orden

```
-- ----------------------------------------------------------------------
-- Ejercicio. Demostrar que si a, b, c, d y e son números reales tales
-- que
--    a ≤ b
--    b < c
--    c ≤ d
--    d < e
-- entonces
--    a < e
-- ----------------------------------------------------------------------

import data.real.basic

variables a b c d e : ℝ

-- 1ª demostración
-- ===============

example
  (h₀ : a ≤ b)
  (h₁ : b < c)
  (h₂ : c ≤ d)
  (h₃ : d < e) :
  a < e :=
begin
  apply lt_of_le_of_lt h₀,
  apply lt_trans h₁,
  apply lt_of_le_of_lt h₂,
  exact h₃,
end

-- El desarrollo de la prueba es
--
--    a b c d e : ℝ,
--    h₀ : a ≤ b,
--    h₁ : b < c,
--    h₂ : c ≤ d,
--    h₃ : d < e
--    ⊢ a < e
-- apply lt_of_le_of_lt h₀,
--    ⊢ b < e
-- apply lt_trans h₁,
--    ⊢ c < e
-- apply lt_of_le_of_lt h₂,
```

```
--    ⊢ d < e
-- exact h₃,
--    no goals

-- 2ª demostración
-- ===============

example
  (h₀ : a ≤ b)
  (h₁ : b < c)
  (h₂ : c ≤ d)
  (h₃ : d < e) :
  a < e :=
calc
  a ≤ b   : h₀
  ... < c : h₁
  ... ≤ d : h₂
  ... < e : h₃

-- 3ª demostración
-- ===============

example
  (h₀ : a ≤ b)
  (h₁ : b < c)
  (h₂ : c ≤ d)
  (h₃ : d < e) :
  a < e :=
by finish
```

- Demostraciones por aritmética lineal

```
-- ----------------------------------------------------------------------
-- Ejercicio 1. Sean a, b, c, d y e números reales. Demostrar que si
--    a ≤ b
--    b < c
--    c ≤ d
--    d < e
-- entonces
--    a < e
-- ----------------------------------------------------------------------

import data.real.basic
```

```
variables a b c d e : ℝ

example
  (h₀ : a ≤ b)
  (h₁ : b < c)
  (h₂ : c ≤ d)
  (h₃ : d < e)
  : a < e :=
by linarith

-- ---------------------------------------------------------------------
-- Ejercicio 2. Demostrar que si
--    2 * a ≤ 3 * b
--    1 ≤ a
--    d = 2
-- entonces
--    d + a ≤ 5 * b
-- ---------------------------------------------------------------------

example
  (h   : 2 * a ≤ 3 * b)
  (h'  : 1 ≤ a)
  (h'' : d = 2)
  : d + a ≤ 5 * b :=
by linarith
```

- Aritmética lineal con argumentos

```
-- ---------------------------------------------------------------------
-- Ejercicio. Sean a, b, y c números reales. Demostrar  que si
--    1 ≤ a
--    b ≤ c
-- entonces
--    2 + a + exp b ≤ 3 * a + exp c
-- ---------------------------------------------------------------------


import analysis.special_functions.exp_log

open real

variables a b c : ℝ

example
```

```
  (h  : 1 ≤ a)
  (h' : b ≤ c)
  : 2 + a + exp b ≤ 3 * a + exp c :=
by linarith [exp_le_exp.mpr h']
```

- Lemas de desigualdades en R

```
-- ---------------------------------------------------------------------
-- Ejercicio. Calcular el tipo de los siguientes lemas
--    exp_le_exp
--    exp_lt_exp
--    log_le_log
--    log_lt_log
--    add_le_add
--    add_lt_add_of_le_of_lt
--    add_nonneg
--    add_pos
--    add_pos_of_pos_of_nonneg
--    exp_pos
-- ---------------------------------------------------------------------

import analysis.special_functions.exp_log

open real

variables  a b c d : ℝ

#check (exp_le_exp : exp a ≤ exp b ↔ a ≤ b)
#check (exp_lt_exp : exp a < exp b ↔ a < b)
#check (log_le_log : 0 < a → 0 < b → (log a ≤ log b ↔ a ≤ b))
#check (log_lt_log : 0 < a → a < b → log a < log b)
#check (add_le_add : a ≤ b → c ≤ d → a + c ≤ b + d)
#check (add_lt_add_of_le_of_lt : a ≤ b → c < d → a + c < b + d)
#check (add_nonneg : 0 ≤ a → 0 ≤ b → 0 ≤ a + b)
#check (add_pos : 0 < a → 0 < b → 0 < a + b)
#check (add_pos_of_pos_of_nonneg : 0 < a → 0 ≤ b → 0 < a + b)
#check (exp_pos : ∀ a, 0 < exp a)
```

- Desigualdad de exponenciales (reescritura con el bicondicional)

```
-- ---------------------------------------------------------------------
-- Ejercicio 1. Realizar las siguientes acciones
--    1. Importar la teoría de exponeciales y logaritmos.
--    2. Abrir la teoría de los reales
--    3. Declarar a y b como variables sobre los reales.
-- ---------------------------------------------------------------------

import analysis.special_functions.exp_log   -- 1
open real                                   -- 2
variables (a b : ℝ)                         -- 3

-- ---------------------------------------------------------------------
-- Ejercicio 2. Calcular el tipo del lema exp_le_exp
-- ---------------------------------------------------------------------

#check @exp_le_exp a b

-- Comentario: Al colocar el cursor sobre check se obtiene
--    exp_le_exp : a.exp ≤ b.exp ↔ a ≤ b

-- ---------------------------------------------------------------------
-- Ejercicio 3. Demostrar que si
--    a ≤ b
--    exp a ≤ exp b
-- ---------------------------------------------------------------------

-- 1ª demostración
example
  (h : a ≤ b)
  : exp a ≤ exp b :=
begin
  rw exp_le_exp,
  exact h
end

-- 2ª demostración
example
  (h : a ≤ b)
  : exp a ≤ exp b :=
exp_le_exp.mpr h

-- Nota: Con mpr se indica en modus pones inverso. Por ejemplo, si
-- h: A ↔ B, entonces h.mpr es B → A y h.mp es A → B
```

- Eliminación de bicondicional

```
-- ---------------------------------------------------------------------
-- Ejercicio 1. Realizar las siguientes acciones
--    1. Importar la teoría de exponeciales y logaritmos.
--    2. Abrir la teoría de los reales
--    3. Declarar a, b, c, d y e como variables sobre los reales.
-- ---------------------------------------------------------------------

import analysis.special_functions.exp_log   -- 1
open real                                   -- 2
variables a b c d e : ℝ                     -- 3

-- ---------------------------------------------------------------------
-- Ejercicio 2. Calcular el tipo de los siguientes lemas
--    add_lt_add_of_le_of_lt
--    exp_lt_exp
--    le_refl
-- ---------------------------------------------------------------------

#check @add_lt_add_of_le_of_lt ℝ _ a b c d
#check @exp_lt_exp a b
#check le_refl

-- Comentario: Colocando el cursor sobre check se obtiene
--    add_lt_add_of_le_of_lt : a ≤ b → c < d → a + c < b + d
--    exp_lt_exp : a.exp < b.exp ↔ a < b
--    le_refl : ∀ (a : ?M_1), a ≤ a

-- ---------------------------------------------------------------------
-- Ejercicio 3. Demostrar que si
--    a ≤ b
--    c < d
-- entonces
--    a + exp c + e < b + exp d + e :=
-- ---------------------------------------------------------------------

example
  (h₀ : a ≤ b)
  (h₁ : c < d)
  : a + exp c + e < b + exp d + e :=
begin
  apply add_lt_add_of_lt_of_le,
  { apply add_lt_add_of_le_of_lt h₀,
    apply exp_lt_exp.mpr h₁, },
  apply le_refl
end
```

```
-- El desarrollo de la prueba es
--
--    a b c d e : ℝ,
--    h₀ : a ≤ b,
--    h₁ : c < d
--    ⊢ a + c.exp + e < b + d.exp + e
-- apply add_lt_add_of_lt_of_le,
-- | { apply add_lt_add_of_le_of_lt h₀,
-- |       ⊢ a + c.exp < b + d.exp
-- |   apply exp_lt_exp.mpr h₁ },
--    ⊢ e ≤ e
-- apply le_refl
--    no_goals
```

- Ejercicio sobre desigualdades

```
-- ---------------------------------------------------------------------
-- Ejercicio 1. Realizar las siguientes acciones
--    1. Importar la teoría de exponeciales y logaritmos.
--    2. Abrir la teoría de los reales
--    3. Declarar a, b, c, d y e como variables sobre los reales.
-- ---------------------------------------------------------------------

import analysis.special_functions.exp_log   -- 1
open real                                   -- 2
variables a b c d e : ℝ                     -- 3

-- ---------------------------------------------------------------------
-- Ejercicio 2. Demostrar que si
--    d ≤ e
-- entonces
--    c + exp (a + d) ≤ c + exp (a + e)
-- ---------------------------------------------------------------------

example
  (h₀ : d ≤ e)
  : c + exp (a + d) ≤ c + exp (a + e) :=
begin
  apply add_le_add,
   apply le_refl,
  apply exp_le_exp.mpr,
  apply add_le_add,
   apply le_refl,
```

```
  exact h₀,
end

-- El desarrollo de la prueba es
--
--    a c d e : ℝ,
--    h₀ : d ≤ e
--    ⊢ c + (a + d).exp ≤ c + (a + e).exp
-- apply add_le_add,
-- |   ⊢ c ≤ c
-- | apply le_refl,
--    ⊢ (a + d).exp ≤ (a + e).exp
-- apply exp_le_exp.mpr,
--    ⊢ a + d ≤ a + e
-- apply add_le_add,
-- |    ⊢ a ≤ a
-- | apply le_refl,
--    ⊢ d ≤ e
-- exact h₀,
--    no goals

-- ---------------------------------------------------------------------
-- Ejercicio 3. Demostrar que
--    0 < 1
-- ---------------------------------------------------------------------

example : (0 : ℝ) < 1 :=
by norm_num

-- Nota: La táctica norm_num normaliza expresiones numéricas. Ver
-- https://bit.ly/3hoJMgQ

-- ---------------------------------------------------------------------
-- Ejercicio 4. Demostrar que si
--    a ≤ b
-- entonces
--    log (1 + exp a) ≤ log (1 + exp b) :=
-- ---------------------------------------------------------------------

example
  (h : a ≤ b)
  : log (1 + exp a) ≤ log (1 + exp b) :=
begin
  have h₀ : 0 < 1 + exp a,
  { apply add_pos,
```

```
    norm_num,
    apply exp_pos, },
  have h₁ : 0 < 1 + exp b,
  { apply add_pos,
    norm_num,
    apply exp_pos },
  apply (log_le_log h₀ h₁).mpr,
  apply add_le_add,
   apply le_refl,
  apply exp_le_exp.mpr h,
end

-- El desarrollo de la prueba es
--
--    a b : ℝ,
--    h : a ≤ b
--    ⊢ (1 + a.exp).log ≤ (1 + b.exp).log
-- have h₀ : 0 < 1 + exp a,
-- |    ⊢ 0 < 1 + a.exp
-- | apply add_pos,
-- |    ⊢ 0 < 1
-- | norm_num,
-- |    ⊢ 0 < a.exp
-- | apply exp_pos },
--    a b : ℝ,
--    h : a ≤ b,
--    h₀ : 0 < 1 + a.exp
--    ⊢ (1 + a.exp).log ≤ (1 + b.exp).log
-- have h₁ : 0 < 1 + exp b,
--    ⊢ 0 < 1 + b.exp
-- | apply add_pos,
-- |    ⊢ 0 < 1
-- | norm_num,
-- |    0 < b.exp
-- | apply exp_pos },
--    a b : ℝ,
--    h : a ≤ b,
--    h₀ : 0 < 1 + a.exp,
--    h₁ : 0 < 1 + b.exp
--    ⊢ (1 + a.exp).log ≤ (1 + b.exp).log
-- apply (log_le_log h₀ h₁).mpr,
--    ⊢ 1 + a.exp ≤ 1 + b.exp
-- apply add_le_add,
-- |    ⊢ 1 ≤ 1
-- | apply le_refl,
```

```
--    ⊢ a.exp ≤ b.exp
-- apply exp_le_exp.mpr h,
--    no goals

-- Comentario. Los lemas empleados son
#check (add_le_add : a ≤ b → c ≤ d → a + c ≤ b + d)
#check (le_refl : ∀ (a : real), a ≤ a)
#check (exp_le_exp : exp a ≤ exp b ↔ a ≤ b)
#check (add_pos : 0 < a → 0 < b → 0 < a + b)
#check (exp_pos : ∀ a, 0 < exp a)
#check (log_le_log : 0 < a → 0 < b → (log a ≤ log b ↔ a ≤ b))
```

- Uso de library_search

```
-- ---------------------------------------------------------------------
-- Ejercicio . Demostrar que, para todo número real a,
--    0 ≤ a^2
-- ---------------------------------------------------------------------

import data.real.basic
import tactic

example (a : ℝ) : 0 ≤ a^2 :=
begin
  -- library_search,
  exact pow_two_nonneg a,
end

-- Notas:
-- + Nota 1: Al colocar el cursor sobre library_search (después de descomentar
--   la línea) escribe el mensaje
--      Try this: exact pow_two_nonneg a
-- + Nota 2: Para usar library_search hay que importar tactic.
```

- Ejercicio con library_search

```
-- ---------------------------------------------------------------------
-- Ejercicio. Sean a, b y c números reales. Demostrar que si
--    a ≤ b
-- entonces
--    c - exp b ≤ c - exp a :=
-- ---------------------------------------------------------------------
```

```
import import analysis.special_functions.exp_log
import tactic

open real

variables a b c : ℝ

-- 1ª demostración
-- ===============

example
  (h : a ≤ b)
  : c - exp b ≤ c - exp a :=
begin
   apply add_le_add,
   apply le_refl,
   apply neg_le_neg,
   apply exp_le_exp.mpr h,
end

-- El desarrollo de la prueba es
--
--    a b c : ℝ,
--    h : a ≤ b
--    ⊢ c - b.exp ≤ c - a.exp
-- apply add_le_add,
-- |    ⊢ c ≤ c
-- | apply le_refl,
--    ⊢ -b.exp ≤ -a.exp
-- apply neg_le_neg,
--    ⊢ a.exp ≤ b.exp
-- apply exp_le_exp.mpr h,
--    no goals

-- 2ª demostración
-- ===============

example
  (h : a ≤ b)
  : c - exp b ≤ c - exp a :=
-- by library_search [exp_le_exp.mpr h]
by exact sub_le_sub_left (exp_le_exp.mpr h) c

-- 3ª demostración
```

```
-- ===============

example
  (h : a ≤ b)
  : c - exp b ≤ c - exp a :=
by linarith [exp_le_exp.mpr h]

-- Los lemas usados son:
#check (add_le_add : a ≤ b → c ≤ d → a + c ≤ b + d)
#check (exp_le_exp : exp a ≤ exp b ↔ a ≤ b)
#check (le_refl : ∀ (a : real), a ≤ a)
#check (neg_le_neg : a ≤ b → -b ≤ -a)
```

- Desigualdades con calc

```
-- ---------------------------------------------------------------------
-- Ejercicio. Sean a y b números reales. Demostrar que
--    2*a*b ≤ a^2 + b^2
-- ---------------------------------------------------------------------

import data.real.basic
import tactic

variables a b : ℝ

-- 1ª demostración
example : 2*a*b ≤ a^2 + b^2 :=
begin
  have h : 0 ≤ a^2 - 2*a*b + b^2,
  calc
    a^2 - 2*a*b + b^2 = (a - b)^2       : by ring
                  ... ≥ 0               : by apply pow_two_nonneg,
  calc
    2*a*b = 2*a*b + 0                   : by ring
      ... ≤ 2*a*b + (a^2 - 2*a*b + b^2) : add_le_add (le_refl _) h
      ... = a^2 + b^2                   : by ring
end

-- 2ª demostración
example : 2*a*b ≤ a^2 + b^2 :=
begin
  have : 0 ≤ a^2 - 2*a*b + b^2,
  calc
    a^2 - 2*a*b + b^2 = (a - b)^2 : by ring
```

```
    ... ≥ 0                          : by apply pow_two_nonneg,
  linarith,
end
```

- Ejercicio desigualdades absolutas

```
-- ---------------------------------------------------------------------
-- Ejercicio. Sean a y b números reales. Demostrar que
--    abs (a*b) ≤ (a^2 + b^2) / 2 :=
-- ---------------------------------------------------------------------

import data.real.basic

variables a b : ℝ

example : abs (a*b) ≤ (a^2 + b^2) / 2 :=
begin
  apply abs_le_of_le_of_neg_le,
  { have h1 : 0 ≤ a^2 - 2*a*b + b^2,
      calc 0 ≤ (a-b)^2                 : by exact pow_two_nonneg (a - b)
         ... = a^2-2*a*b+b^2           : by ring,
    have h2 : 2*(a*b) ≤ a^2 + b^2,
      calc 2*(a*b)
           ≤ 2*(a*b)+(a^2-2*a*b+b^2)   : by exact le_add_of_nonneg_right h1
       ... = a^2 + b^2                 : by ring,
    by linarith [h2] },
  { have h3 : 0 ≤ a^2 + 2*a*b + b^2,
      calc 0 ≤ (a+b)^2                 : by exact pow_two_nonneg (a + b)
         ... = a^2+2*a*b+b^2           : by ring,
    have h4 : -2*(a*b) ≤ a^2 + b^2,
      calc -2*(a*b)
           ≤ -2*(a*b)+(a^2+2*a*b+b^2)  : by exact le_add_of_nonneg_right h3
       ... = a^2 + b^2                 : by ring,
    by linarith [h4] },
end
```

2.4. Más sobre orden y divisibilidad

2.4.1. Mínimos y máximos

- Caracterización del mínimo

```
-- ---------------------------------------------------------------------
-- Ejercicio. Sean a, b y c números reales. Calcular los tipos de
--    min_le_left a b
--    min_le_right a b
--    @le_min ℝ _ a b c
-- ---------------------------------------------------------------------

import data.real.basic

variables a b c : ℝ

#check min_le_left a b
#check min_le_right a b
#check @le_min ℝ _ a b c

-- Comentario al colocar el cursor sobre check se obtiene
--    min_le_left a b : min a b ≤ a
--    min_le_right a b : min a b ≤ b
--    le_min : c ≤ a → c ≤ b → c ≤ min a b
```

- Caracterización del máximo

```
-- ---------------------------------------------------------------------
-- Ejercicio. Sean a, b y c números reales. Calcular los tipos de
--    le_max_left a b
--    le_max_right a b
--    @max_le ℝ _ a b c
-- ---------------------------------------------------------------------

import data.real.basic

variables a b c : ℝ

#check le_max_left a b
#check le_max_right a b
#check @max_le ℝ _ a b c

-- Comentario al colocar el cursor sobre check se obtiene
--    le_max_left a b : a ≤ max a b
--    le_max_right a b : b ≤ max a b
--    max_le : a ≤ c → b ≤ c → max a b ≤ c
```

- Conmutatividad del mínimo

```
-- ----------------------------------------------------------------------
-- Ejercicio. Sean a y b números reales. Demostrar que
--    min a b = min b a
-- ----------------------------------------------------------------------

import data.real.basic

variables a b : ℝ

-- 1ª demostración
-- ===============

example : min a b = min b a :=
begin
  apply le_antisymm,
  { show min a b ≤ min b a,
    apply le_min,
    { apply min_le_right },
    apply min_le_left },
  { show min b a ≤ min a b,
    apply le_min,
    { apply min_le_right },
    apply min_le_left }
end

-- Nota: Se usa "show" para indicar lo que se demuestra en cada bloque.

-- El desarrollo de la prueba es
--
--    ⊢ min a b = min b a
-- apply le_antisymm,
-- |    ⊢ min a b ≤ min b a
-- | apply le_min,
-- | |    ⊢ min a b ≤ b
-- | | { apply min_le_right },
-- | |    ⊢ min a b ≤ a
-- | apply min_le_left,
-- |    ⊢ min b a ≤ min a b
-- | apply le_min,
-- | |    ⊢ min b a ≤ a
-- | { apply min_le_right },
-- | |    ⊢ min b a ≤ b
-- | apply min_le_left }
--    no goals
```

```
-- 2ª demostración (con lema local)
-- ================================

example : min a b = min b a :=
begin
  have h : ∀ x y, min x y ≤ min y x,
  { intros x y,
    apply le_min,
    apply min_le_right,
    apply min_le_left },
  apply le_antisymm, apply h, apply h
end

-- Nota: La táctica "intros" introduce las variables en el contexto. Ver
-- https://bit.ly/2UF1EdL

-- 3ª demostración (con repeat)
-- ============================

example : min a b = min b a :=
begin
  apply le_antisymm,
  repeat {
    apply le_min,
    apply min_le_right,
    apply min_le_left },
end

-- Nota. La táctica "repeat" aplica una táctica recursivamente a todos los
-- subobjetivos. Ver https://bit.ly/2YuO5P9

-- Lemas usados
-- ============

#check (le_antisymm : a ≤ b → b ≤ a → a = b)
#check (le_min : c ≤ a → c ≤ b → c ≤ min a b)
#check (min_le_left a b : min a b ≤ a)
#check (min_le_right a b : min a b ≤ b)
```

▪ Conmutatividad del máximo

```
-- ---------------------------------------------------------------------
-- Ejercicio. Sean a y b números reales. Demostrar que
--    max a b = max b a
```

```
-- ----------------------------------------------------------------------

import data.real.basic

variables a b : ℝ

-- 1ª demostración
-- ===============

example : max a b = max b a :=
begin
  apply le_antisymm,
  { show max a b ≤ max b a,
    apply max_le,
    { apply le_max_right },
    apply le_max_left },
  { show max b a ≤ max a b,
    apply max_le,
    { apply le_max_right },
    apply le_max_left }
end

-- 2ª demostración
-- ===============

example : max a b = max b a :=
begin
  have h : ∀ x y, max x y ≤ max y x,
  { intros x y,
    apply max_le,
    apply le_max_right,
    apply le_max_left },
  apply le_antisymm, apply h, apply h
end

-- 3ª demostración
-- ===============

example : max a b = max b a :=
begin
  apply le_antisymm,
  repeat {
    apply max_le,
    apply le_max_right,
    apply le_max_left },
```

```
end

-- Lemas usados
-- ============

#check (le_max_left a b : a ≤ max a b)
#check (le_max_right a b : b ≤ max a b)
#check (max_le : a ≤ c → b ≤ c → max a b ≤ c)
```

- Ejercicio: Asociatividad del mínimo

```
-- ---------------------------------------------------------------------
-- Ejercicio. Sean a, b y c números reales. Demostrar que
--    min (min a b) c = min a (min b c)
-- ---------------------------------------------------------------------

import data.real.basic tactic

variables a b c : ℝ

-- Se usará el siguiente lema auxilar.
lemma aux1 : min (min a b) c ≤ min a (min b c) :=
begin
  apply le_min,
  { show min (min a b) c ≤ a,
      calc min (min a b) c ≤ min a b : by apply min_le_left
                       ...  ≤ a      : by apply min_le_left },
  { show min (min a b) c ≤ min b c,
    apply le_min,
    { show min (min a b) c ≤ b,
        calc  min (min a b) c ≤  min a b : by apply min_le_left
                            ... ≤ b      : by apply min_le_right },
    { show min (min a b) c ≤ c,
      { apply min_le_right }}},
end

example : min (min a b) c = min a (min b c) :=
begin
  apply le_antisymm,
  { show min (min a b) c ≤ min a (min b c),
      by exact aux1 a b c },
  { show min a (min b c) ≤ min (min a b) c,
      calc min a (min b c) = min (min b c) a : by apply min_comm
                       ... = min (min c b) a : by {congr' 1, apply min_comm}
```

```
                      ... ≤ min c (min b a) : by apply aux1
                      ... = min c (min a b) : by {congr' 1, apply min_comm}
                      ... = min (min a b) c : by apply min_comm
  },
end
```

- Ejercicio: Mínimo de suma

```
-- ---------------------------------------------------------------------
-- Ejercicio. Sean a, b y c números reales. Demostrar que
--    min a b + c = min (a + c) (b + c)
-- ---------------------------------------------------------------------

import data.real.basic

variables a b c : ℝ

example : min a b + c = min (a + c) (b + c) :=
begin
  by_cases (a ≤ b),
  { have h1 : a + c ≤ b + c,
      apply add_le_add_right h,
    calc min a b + c = a + c               : by simp [min_eq_left h]
                 ... = min (a + c) (b + c) : by simp [min_eq_left h1]},
  { have h2: b ≤ a,
      linarith,
    have h3 : b + c ≤ a + c,
      { exact add_le_add_right h2 c },
    calc min a b + c = b + c               : by simp [min_eq_right h2]
                 ... = min (a + c) (b + c) : by simp [min_eq_right h3]},
end
```

- Lema abs_add

```
-- ---------------------------------------------------------------------
-- Ejercicio. Calcular el tipo de
--     abs_add
-- ---------------------------------------------------------------------

import data.real.basic

#check abs_add
```

```
-- Comentario: Colando el cursor sobre check se obtiene
--    abs_add : ∀ (a b : ?M_1), abs (a + b) ≤ abs a + abs b
```

- Ejercicio: abs_sub

```
-- ---------------------------------------------------------------------
-- Ejercicio. Sean a y b números reales. Demostrar que
--    abs a - abs b ≤ abs (a - b)
-- ---------------------------------------------------------------------

import data.real.basic

variables a b : ℝ

example : abs a - abs b ≤ abs (a - b) :=
begin
  apply sub_le_iff_le_add.mpr,
  have h1 : abs a = abs ((a - b) + b),
    by ring,
  rw h1,
  apply abs_add,
end

-- Lemas usados:
#check (abs_add : ∀ a b : ℝ, abs (a + b) ≤ abs a + abs b)
#check (sub_le_iff_le_add : a - b ≤ c ↔ a ≤ c + b)
```

2.4.2. Divisibilidad

- Propiedades de divisibilidad

```
-- ---------------------------------------------------------------------
-- Ejercicio 1. Realizar la siguientes acciones:
--    1. Importar la teoría de mcd sobre los naturales.
--    2. Declarar x, y y z como variables sobre los naturales.
-- ---------------------------------------------------------------------

import data.nat.gcd   -- 1
variables x y z : ℕ   -- 2

-- ---------------------------------------------------------------------
-- Ejercicio 2. Demostrar que si
```

```
--    x | y
--    y | z
-- entonces
--    x | z
-- ----------------------------------------------------------------------

example
  (h₀ : x | y)
  (h₁ : y | z)
  : x | z :=
dvd_trans h₀ h₁

-- ----------------------------------------------------------------------
-- Ejercicio 3. Demostrar que
--    x | y * x * z
-- ----------------------------------------------------------------------

example : x | y * x * z :=
begin
  apply dvd_mul_of_dvd_left,
  apply dvd_mul_left
end

-- Su desarrollo es
--
--    ⊢ x | y * x * z
-- apply dvd_mul_of_dvd_left,
--    ⊢ x | y * x
-- apply dvd_mul_left
--    no goals

-- ----------------------------------------------------------------------
-- Ejercicio 4. Demostrar que
--    x | x^2
-- ----------------------------------------------------------------------

example : x | x^2 :=
begin
  rw nat.pow_two,
  apply dvd_mul_left
end

-- Su desarrollo es
--
```

```
--    ⊢ x | x ^ 2
-- rw nat.pow_two,
--    ⊢ x | x * x
-- apply dvd_mul_left
--    no goals

-- Lemas usados
-- ============

#check (dvd_trans : x ∣ y → y ∣ z → x ∣ z)
#check (dvd_mul_of_dvd_left : x ∣ y → ∀ (c : ℕ), x ∣ y * c)
#check (dvd_mul_left : ∀ (a b : ℕ), a ∣ b * a)
#check (nat.pow_two : ∀ (a : ℕ), a ^ 2 = a * a)
```

- Ejercicio de divisibilidad

```
-- ---------------------------------------------------------------------
-- Ejercicio. Demostrar que si
--    x | w
-- entonces
--    x | y * (x * z) + x^2 + w^2
-- ---------------------------------------------------------------------

import data.nat.gcd

variables w x y z : ℕ

example
  (h : x ∣ w)
  : x ∣ y * (x * z) + x^2 + w^2 :=
begin
  apply dvd_add,
    apply dvd_add,
      apply dvd_mul_of_dvd_right,
      apply dvd_mul_right,
    rw nat.pow_two,
    apply dvd_mul_right,
  rw nat.pow_two,
  apply dvd_mul_of_dvd_left h,
end

-- Su desarrollo es
--
```

```
-- ⊢ x ∣ y * (x * z) + x ^ 2 + w ^ 2
--     apply dvd_add,
-- ⊢ x ∣ y * (x * z) + x ^ 2
-- |     apply dvd_add,
-- | ⊢ x ∣ y * (x * z)
-- | |     apply dvd_mul_of_dvd_right,
-- | | ⊢ x ∣ x * z
-- | |     apply dvd_mul_right,
-- | ⊢ x ∣ x ^ 2
-- | |     rw nat.pow_two,
-- | | ⊢ x ∣ x * x
-- | |     apply dvd_mul_right,
-- ⊢ x ∣ w ^ 2
-- |     rw nat.pow_two,
-- | ⊢ x ∣ w * w
-- |     apply dvd_mul_of_dvd_left h,
-- no goals

-- Lemas usados
-- ============

#check (dvd_add : x ∣ y → x ∣ z → x ∣ y + z)
#check (dvd_mul_of_dvd_left : x ∣ y → ∀ (c : ℕ), x ∣ y * c)
#check (dvd_mul_of_dvd_right : x ∣ y → ∀ (c : ℕ), x ∣ c * y)
#check (dvd_mul_right : ∀ (a b : ℕ), a ∣ a * b)
#check (nat.pow_two : ∀ (a : ℕ), a ^ 2 = a * a)
```

- Propiedades de gcd y lcm

```
-- ---------------------------------------------------------------------
-- Ejercicio. Calcular el tipo de los siguientes lemas
--    gcd_zero_right
--    gcd_zero_left
--    lcm_zero_right
--    lcm_zero_left
-- ---------------------------------------------------------------------

import data.nat.gcd

open nat

variables n : ℕ
```

```
#check (gcd_zero_right n : gcd n 0 = n)
#check (gcd_zero_left n  : gcd 0 n = n)
#check (lcm_zero_right n : lcm n 0 = 0)
#check (lcm_zero_left n  : lcm 0 n = 0)
```

- Conmutatividad del gcd

```
-- ---------------------------------------------------------------------
-- Ejercicio. Demostrar que
--    gcd m n = gcd n m
-- ---------------------------------------------------------------------

import data.nat.gcd

open nat

variables k m n : ℕ

example : gcd m n = gcd n m :=
begin
  apply dvd_antisymm,
  { apply dvd_gcd (gcd_dvd_right m n) (gcd_dvd_left m n)},
  { apply dvd_gcd (gcd_dvd_right n m) (gcd_dvd_left n m)},
end

-- Su desarrollo es
--
-- ⊢ m.gcd n = n.gcd m
--    apply dvd_antisymm,
-- ⊢ m.gcd n ∣ n.gcd m
-- |    apply dvd_gcd (gcd_dvd_right m n) (gcd_dvd_left m n)
-- ⊢ n.gcd m ∣ m.gcd n
-- |    apply dvd_gcd (gcd_dvd_right n m) (gcd_dvd_left n m)

-- Lemas usados
-- ============

#check (dvd_antisymm : m ∣ n → n ∣ m → m = n)
#check (dvd_gcd : k ∣ m → k ∣ n → k ∣ gcd m n)
#check (gcd_dvd_left : ∀ (m n : ℕ), gcd m n ∣ m)
#check (gcd_dvd_right : ∀ (m n : ℕ), gcd m n ∣ n)
```

2.5. Demostraciones sobre estructuras algebraicas

2.5.1. Órdenes

- Órdenes parciales

```
-- ----------------------------------------------------------------------
-- Ejercicio 1. Realizar las siguientes acciones:
--    1. Importar la teoría de órdenes
--    2. Declarar α como un tipo sobre los órdenes parciales
--    3. x, y y z como variables sobre α.
-- ----------------------------------------------------------------------

import order.basic                          -- 1
variables {α : Type*} [partial_order α]     -- 2
variables x y z : α                         -- 3

-- ----------------------------------------------------------------------
-- Ejercicio 2. Calcular los tipos de las siguientes expresiones
--    x ≤ y
--    le_refl x
--    @le_trans α _ x y z
-- ----------------------------------------------------------------------

#check x ≤ y
#check le_refl x
#check @le_trans α _ x y z

-- Conentario: Al colocar el cursor sobre check se obtiene
--    x ≤ y : Prop
--    le_refl x : x ≤ x)
--    le_trans : x ≤ y → y ≤ z → x ≤ z)

-- Nota: Las letras griegas se escriben con \a, \b, ...
```

- Orden estricto

```
-- ----------------------------------------------------------------------
-- Ejercicio 1. Realizar las siguientes acciones:
--    1. Importar la teoría de órdenes
--    2. Declarar α como un tipo sobre los órdenes parciales
```

```
--    3. x, y y z como variables sobre α.
-- ----------------------------------------------------------------------

import order.basic                          -- 1
variables {α : Type*} [partial_order α]   -- 2
variables x y z : α                         -- 3

-- ----------------------------------------------------------------------
-- Ejercicio 2. Calcular el tipo de las siguientes expresiones
--    x < y
--    lt_irrefl x
--    @lt_trans α _ x y z
--    @lt_of_le_of_lt α _ x y z
--    @lt_of_lt_of_le α _ x y z
--    @lt_iff_le_and_ne  α _ x y
-- ----------------------------------------------------------------------

#check x < y
#check lt_irrefl x
#check @lt_trans α _ x y z
#check @lt_of_le_of_lt α _ x y z
#check @lt_of_lt_of_le α _ x y z
#check @lt_iff_le_and_ne  α _ x y

-- Comentario: Al colocar el cursor sobre check se obtiene
--    x < y : Prop
--    lt_irrefl x : ¬ x < x
--    lt_trans : x < y → y < z → x < z
--    lt_of_le_of_lt : x ≤ y → y < z → x < z
--    lt_of_lt_of_le : x < y → y ≤ z → x < z
--    lt_iff_le_and_ne : x < y ↔ x ≤ y ∧ x ≠ y
```

2.5.2. Retículos

- Retículos

```
-- ----------------------------------------------------------------------
-- Ejercicio 1. Realizar las siguientes acciones:
--    1. Importar la teoría de retículos
--    2. Declarar α como un tipo sobre los retículos.
--    3. x, y y z como variables sobre α.
-- ----------------------------------------------------------------------
```

```
import order.lattice                -- 1
variables {α : Type*} [lattice α]   -- 2
variables x y z : α                 -- 3

-- ----------------------------------------------------------------------
-- Ejercicio 2. Calcular el tipo de las siguientes expresiones
--    x ⊓ y
--    @inf_le_left α _ x y
--    @inf_le_right α _ x y
--    @le_inf α _ z x y
--
--    x ⊔ y
--    @le_sup_left α _ x y
--    @le_sup_right α _ x y
--    @sup_le α _ x y z
-- ----------------------------------------------------------------------

#check x ⊓ y
#check @inf_le_left α _ x y
#check @inf_le_right α _ x y
#check @le_inf α _ z x y

#check x ⊔ y
#check @le_sup_left α _ x y
#check @le_sup_right α _ x y
#check @sup_le α _ x y z


-- Comentarios:
-- 1. Para ver cómo se escribe un símbolo, se coloca el cursor sobre el
--    símbolo y se presiona C-c C-k
-- 2. El ínfimo ⊓ se escribe con \glb de "greatest lower bound"
-- 3. El supremo ⊔ se escribe con \lub de "least upper bound"
-- 4. En mathlib se unsa inf o sup para los nombres sobre ínfimo o supremo.
-- 5. Al colocar el cursor sobre check se obtiene
--       x ⊓ y : α
--       inf_le_left : x ⊓ y ≤ x
--       inf_le_right : x ⊓ y ≤ y
--       le_inf : z ≤ x → z ≤ y → z ≤ x ⊓ y
--       x ⊔ y : α
--       le_sup_left : x ≤ x ⊔ y
--       le_sup_right: y ≤ x ⊔ y
--       sup_le : x ≤ z → y ≤ z → x ⊔ y ≤ z
```

- Conmutatividad del ínfimo

```
-- ---------------------------------------------------------------------
-- Ejercicio. Demostrar que en los retículos se verifica que
--     x ⊓ y = y ⊓ x
-- ---------------------------------------------------------------------

import order.lattice

variables {α : Type*} [lattice α]
variables x y : α

-- 1ª demostración
-- ===============

lemma aux : x ⊓ y ≤ y ⊓ x :=
begin
  apply le_inf,
  apply inf_le_right,
  apply inf_le_left,
end

-- Su desarrollo es
--
-- ⊢ x ⊓ y ≤ y ⊓ x
--    apply le_inf,
-- ⊢ x ⊓ y ≤ y
-- |   apply inf_le_right,
-- ⊢ x ⊓ y ≤ y
-- |   apply inf_le_left,
-- no goals

example : x ⊓ y = y ⊓ x :=
begin
  apply le_antisymm,
  apply aux,
  apply aux,
end

-- Su desarrollo es
--
-- ⊢ x ⊓ y = y ⊓ x
--    apply le_antisymm,
-- ⊢ x ⊓ y ≤ y ⊓ x
-- |   apply aux,
-- ⊢ y ⊓ x ≤ x ⊓ y
-- |   apply aux,
```

```
-- no goals

-- 2ª demostración
-- ===============

example : x ⊓ y = y ⊓ x :=
by apply le_antisymm; simp

-- 3ª demostración
-- ===============

example : x ⊓ y = y ⊓ x :=
inf_comm

-- Lemas usados
-- ============

#check (inf_comm : x ⊓ y = y ⊓ x)
#check (inf_le_left : x ⊓ y ≤ x)
#check (inf_le_right : x ⊓ y ≤ y)
#check (le_antisymm : x ≤ y → y ≤ x → x = y)
#check (le_inf : z ≤ x → z ≤ y → z ≤ x ⊓ y)
```

- Conmutatividad del supremo

```
-- ---------------------------------------------------------------------
-- Ejercicio. Demostrar que en los retículos se verifica que
--     x ⊔ y = y ⊔ x
-- ---------------------------------------------------------------------

import order.lattice

variables {α : Type*} [lattice α]
variables x y z : α

-- 1ª demostración
-- ===============

lemma aux : x ⊔ y ≤ y ⊔ x :=
begin
  apply sup_le,
  apply le_sup_right,
  apply le_sup_left,
```

```
end

-- Su desarrollo es
--
-- ⊢ x ⊔ y ≤ y ⊔ x
--     apply sup_le,
-- ⊢ x ≤ y ⊔ x
-- |   apply le_sup_right,
-- ⊢ y ≤ y ⊔ x
-- |   apply le_sup_left,
-- no goals

example : x ⊔ y = y ⊔ x :=
begin
  apply le_antisymm,
  apply aux,
  apply aux,
end

-- Su desarrollo es
--
-- ⊢ x ⊔ y = y ⊔ x
--     apply le_antisymm,
-- ⊢ x ⊔ y ≤ y ⊔ x
-- |   apply aux,
-- ⊢ y ⊔ x ≤ x ⊔ y
-- |   apply aux,
-- no goals

-- 2ª demostración
-- ===============

example : x ⊔ y = y ⊔ x :=
by apply le_antisymm; simp

-- 3ª demostración
-- ===============

example : x ⊔ y = y ⊔ x :=
sup_comm

-- Lemas usados
-- ============

#check (sup_comm : x ⊔ y = y ⊔ x)
```

```
#check (le_sup_left : x ≤ x ⊔ y)
#check (le_sup_right : y ≤ x ⊔ y)
#check (le_antisymm : x ≤ y → y ≤ x → x = y)
#check (sup_le : x ≤ z → y ≤ z → x ⊔ y ≤ z)
```

- Asociatividad del ínfimo

```
-- ---------------------------------------------------------------------
-- Ejercicio. Demostrar que en los retículos se verifica que
--     (x ⊓ y) ⊓ z = x ⊓ (y ⊓ z)
-- ---------------------------------------------------------------------

import order.lattice

variables {α : Type*} [lattice α]
variables x y z : α

-- 1ª demostración
-- ===============

example : (x ⊓ y) ⊓ z = x ⊓ (y ⊓ z) :=
begin
  apply le_antisymm,
  { apply le_inf,
    { apply inf_le_left_of_le inf_le_left, },
    { apply le_inf (inf_le_left_of_le inf_le_right) inf_le_right}},
  {apply le_inf,
    { apply le_inf inf_le_left (inf_le_right_of_le inf_le_left), },
    { apply inf_le_right_of_le inf_le_right, },},
end

-- Su desarrollo es
--
-- ⊢ x ⊓ y ⊓ z = x ⊓ (y ⊓ z)
--    apply le_antisymm,
-- ⊢ x ⊓ y ⊓ z ≤ x ⊓ (y ⊓ z)
-- |   { apply le_inf,
-- | ⊢ x ⊓ y ⊓ z ≤ x
-- | |     { apply inf_le_left_of_le inf_le_left },
-- | ⊢ x ⊓ y ⊓ z ≤ y ⊓ z
-- | |     { apply le_inf (inf_le_left_of_le inf_le_right) inf_le_right}},
-- ⊢ x ⊓ (y ⊓ z) ≤ x ⊓ y ⊓ z
-- |   {apply le_inf,
```

```
-- | ⊢ x ⊓ (y ⊓ z) ≤ x ⊓ y
-- | |     { apply le_inf inf_le_left (inf_le_right_of_le inf_le_left)},
-- | ⊢ x ⊓ (y ⊓ z) ≤ z
--       { apply inf_le_right_of_le inf_le_right, },},
-- no goals

-- 2ª demostración
-- ===============

example : (x ⊓ y) ⊓ z = x ⊓ (y ⊓ z) :=
le_antisymm
  (le_inf
    (inf_le_left_of_le inf_le_left)
    (le_inf (inf_le_left_of_le inf_le_right) inf_le_right))
  (le_inf
    (le_inf inf_le_left (inf_le_right_of_le inf_le_left))
    (inf_le_right_of_le inf_le_right))

-- 3ª demostración
-- ===============

example : (x ⊓ y) ⊓ z = x ⊓ (y ⊓ z) :=
inf_assoc
```

- Asociatividad del supremo

```
-- ---------------------------------------------------------------------
-- Ejercicio. Demostrar que en los retículos se verifica que
--    x ⊔ y) ⊔ z = x ⊔ (y ⊔ z)
-- ---------------------------------------------------------------------

import order.lattice

variables {α : Type*} [lattice α]
variables x y z : α

-- 1ª demostración
-- ===============

example : (x ⊔ y) ⊔ z = x ⊔ (y ⊔ z) :=
begin
  apply le_antisymm,
  { apply sup_le,
    { apply sup_le le_sup_left (le_sup_right_of_le le_sup_left)},
```

```
    { apply le_sup_right_of_le le_sup_right}},
  { apply sup_le,
    { apply le_sup_left_of_le le_sup_left},
    { apply sup_le (le_sup_left_of_le le_sup_right) le_sup_right}},
end

-- Su desarrollo es
--
-- ⊢ x ⊔ y ⊔ z = x ⊔ (y ⊔ z)
--    apply le_antisymm,
-- | ⊢ x ⊔ y ⊔ z ≤ x ⊔ (y ⊔ z)
-- |    { apply sup_le,
-- | | ⊢ x ⊔ y ≤ x ⊔ (y ⊔ z)
-- | |     { apply sup_le le_sup_left (le_sup_right_of_le le_sup_left)},
-- | | ⊢ z ≤ x ⊔ (y ⊔ z)
-- | |     { apply le_sup_right_of_le le_sup_right}},
-- | ⊢ x ⊔ (y ⊔ z) ≤ x ⊔ y ⊔ z
-- |    { apply sup_le,
-- | | ⊢ x ≤ x ⊔ y ⊔ z
-- | |      { apply le_sup_left_of_le le_sup_left},
-- | | ⊢ y ⊔ z ≤ x ⊔ y ⊔ z
-- | |      { apply sup_le (le_sup_left_of_le le_sup_right) le_sup_right}},
-- no goals

-- 2ª demostración
-- ===============

example : (x ⊔ y) ⊔ z = x ⊔ (y ⊔ z) :=
le_antisymm
  (sup_le
    (sup_le le_sup_left (le_sup_right_of_le le_sup_left))
    (le_sup_right_of_le le_sup_right))
  (sup_le
    (le_sup_left_of_le le_sup_left)
    (sup_le (le_sup_left_of_le le_sup_right) le_sup_right))

-- 3ª demostración
-- ===============

example : x ⊔ y ⊔ z = x ⊔ (y ⊔ z) :=
sup_assoc
```

- Leyes de absorción

```
-- ----------------------------------------------------------------------
-- Ejercicio 1. Realizar las siguientes acciones
--    1. Importar la teoría de retículos.
--    2. Declarar α como un tipo sobre retículos
--    3. Declarar x e y como variabkes sobre α
-- ----------------------------------------------------------------------

import order.lattice                -- 1
variables {α : Type*} [lattice α]   -- 2
variables x y : α                   -- 3

-- ----------------------------------------------------------------------
-- Ejercicio 2. Demostrar que
--    x ⊓ (x ⊔ y) = x
-- ----------------------------------------------------------------------

-- 1ª demostración
-- ===============

example : x ⊓ (x ⊔ y) = x :=
begin
  apply le_antisymm,
  { apply inf_le_left },
  { apply le_inf,
    { apply le_refl },
    { apply le_sup_left }},
end

-- Su desarrollo es
--
-- ⊢ x ⊓ (x ⊔ y) = x
--    apply le_antisymm,
-- | ⊢ x ⊓ (x ⊔ y) ≤ x
-- | |   { apply inf_le_left },
-- | ⊢ x ≤ x ⊓ (x ⊔ y)
-- |   { apply le_inf,
-- | | ⊢ x ≤ x
-- | |     { apply le_refl },
-- | | ⊢ x ≤ x ⊔ y
-- | |     { apply le_sup_left }},
-- no goals

-- 2ª demostración
-- ===============
```

```
example : x ⊓ (x ⊔ y) = x :=
by simp

-- 3ª demostración
example : x ⊓ (x ⊔ y) = x :=
inf_sup_self

-- ----------------------------------------------------------------------
-- Ejercicio 3. Demostrar que
--    x ⊔ (x ⊓ y) = x
-- ----------------------------------------------------------------------

-- 1ª demostración
-- ===============

example : x ⊔ (x ⊓ y) = x :=
begin
  apply le_antisymm,
  { apply sup_le,
    { apply le_refl },
    { apply inf_le_left }},
  { apply le_sup_left },
end

-- Su desarrollo es
--
-- ⊢ x ⊔ x ⊓ y = x
--    apply le_antisymm,
-- | ⊢ x ⊔ x ⊓ y ≤ x
-- |    { apply sup_le,
-- | | ⊢ x ≤ x
-- | |      { apply le_refl },
-- | | ⊢ x ⊓ y ≤ x
-- | |      { apply inf_le_left }},
-- | ⊢ x ≤ x ⊔ x ⊓ y
-- | |    { apply le_sup_left },
-- no goals

-- 2ª demostración
-- ===============

example : x ⊔ (x ⊓ y) = x :=
by simp

-- 3ª demostración
```

```
-- ===============

example : x ⊔ (x ⊓ y) = x :=
sup_inf_self
```

- Retículos distributivos

```
-- ----------------------------------------------------------------------
-- Ejercicio 1. Realizar las siguientes acciones:
--    1. Importar la teoría de retículos
--    2. Declarar α como un tipo sobre los retículos.
--    3. x, y y z como variables sobre α.
-- ----------------------------------------------------------------------

import order.lattice                          -- 1
variables {α : Type*} [distrib_lattice α]     -- 2
variables x y z : α                           -- 3

-- ----------------------------------------------------------------------
-- Ejercicio 2. Calcular el tipo de las siguientes expresiones
--    @inf_sup_left α _ x y z
--    @inf_sup_right α _ x y z
--    @sup_inf_left α _ x y z
--    @sup_inf_right α _ x y z
-- ----------------------------------------------------------------------

#check @inf_sup_left α _ x y z
#check @inf_sup_right α _ x y z
#check @sup_inf_left α _ x y z
#check @sup_inf_right α _ x y z

-- Comentario: Al situar el cursor sobre check se obtiene
--    inf_sup_left  : x ⊓ (y ⊔ z) = (x ⊓ y) ⊔ (x ⊓ z)
--    inf_sup_right : (x ⊔ y) ⊓ z = (x ⊓ z) ⊔ (y ⊓ z)
--    sup_inf_left  : x ⊔ (y ⊓ z) = (x ⊔ y) ⊓ (x ⊔ z)
--    sup_inf_right : (x ⊓ y) ⊔ z = (x ⊔ z) ⊓ (y ⊔ z)
```

- Propiedades distributivas

```
-- ----------------------------------------------------------------------
-- Ejercicio 1. Realizar las siguientes acciones
--    1. Importar la teoría de retículos.
--    2. Declarar α como un tipo sobre retículos
```

```
--    3. Declarar a, b y c como variabkes sobre α
-- ---------------------------------------------------------------------

import order.lattice                -- 1
variables {α : Type*} [lattice α]   -- 2
variables a b c : α                   -- 3

-- ---------------------------------------------------------------------
-- Ejercicio 2. Demostrar que si
--    ∀ x y z : α, x ⊓ (y ⊔ z) = (x ⊓ y) ⊔ (x ⊓ z))
-- entonces
--    (a ⊔ b) ⊓ c = (a ⊓ c) ⊔ (b ⊓ c)
-- ---------------------------------------------------------------------

example
  (h : ∀ x y z : α, x ⊓ (y ⊔ z) = (x ⊓ y) ⊔ (x ⊓ z))
  : (a ⊔ b) ⊓ c = (a ⊓ c) ⊔ (b ⊓ c) :=
calc
  (a ⊔ b) ⊓ c = c ⊓ (a ⊔ b)       : by rw inf_comm
          ... = (c ⊓ a) ⊔ (c ⊓ b) : by rw h
          ... = (a ⊓ c) ⊔ (c ⊓ b) : by rw [@inf_comm _ _ c a]
          ... = (a ⊓ c) ⊔ (b ⊓ c) : by rw [@inf_comm _ _ c b]

-- ---------------------------------------------------------------------
-- Ejercicio 3. Demostrar que si
--    ∀ x y z : α, x ⊔ (y ⊓ z) = (x ⊔ y) ⊓ (x ⊔ z)
-- entonces
--    (a ⊓ b) ⊔ c = (a ⊔ c) ⊓ (b ⊔ c)
-- ---------------------------------------------------------------------

example
  (h : ∀ x y z : α, x ⊔ (y ⊓ z) = (x ⊔ y) ⊓ (x ⊔ z))
  : (a ⊓ b) ⊔ c = (a ⊔ c) ⊓ (b ⊔ c) :=
calc
  (a ⊓ b) ⊔ c = c ⊔ (a ⊓ b)       : by rw sup_comm
          ... = (c ⊔ a) ⊓ (c ⊔ b) : by rw h
          ... = (a ⊔ c) ⊓ (c ⊔ b) : by rw [@sup_comm _ _ c a]
          ... = (a ⊔ c) ⊓ (b ⊔ c) : by rw [@sup_comm _ _ c b]
```

- Anillos ordenados

```
-- ----------------------------------------------------------------------
-- Ejercicio 1. Realizar las siguientes acciones
--    1. Importar la teoría de los anillos ordenados.
--    2. Declarar R como un tipo sobre los anillos ordenados.
--    3. Declarar a y b como variables sobre R.
-- ----------------------------------------------------------------------

import algebra.ordered_ring              -- 1
variables {R : Type*} [ordered_ring R]   -- 2
variables a b : R                        -- 3

-- ----------------------------------------------------------------------
-- Ejercicio 2. Calcular el tipo de las siguientes expresiones
--    @add_le_add_left R _ a b
--    @mul_pos R _ a b
--    zero_ne_one
--    @mul_nonneg R _ a b
-- ----------------------------------------------------------------------

#check @add_le_add_left R _ a b
#check @mul_pos R _ a b
#check zero_ne_one
#check @mul_nonneg R _ a b

-- Comentario: Al colocar el cursor sobre check se obtiene
--    add_le_add_left : a ≤ b → ∀ c, c + a ≤ c + b
--    mul_pos : 0 < a → 0 < b → 0 < a * b
--    zero_ne_one : 0 ≠ 1
--    mul_nonneg : 0 ≤ a → 0 ≤ b → 0 ≤ a * b
```

2.5.3. Anillos ordenados

- Ejercicio sobre anillos ordenados

```
-- ----------------------------------------------------------------------
-- Ejercicio 1. Realizar las siguientes acciones
--    1. Importar la teoría de los anillos ordenados.
--    2. Declarar R como un tipo sobre los anillos ordenados.
--    3. Declarar a, b y c como variables sobre R.
-- ----------------------------------------------------------------------

import algebra.ordered_ring              -- 1
variables {R : Type*} [ordered_ring R]   -- 2
```

```
variables a b c: R                          -- 3

-- ---------------------------------------------------------------------
-- Ejercicio 2. Demostrar que
--    a ≤ b → 0 ≤ b - a
-- ---------------------------------------------------------------------

example : a ≤ b → 0 ≤ b - a :=
begin
  intro h,
  calc
    0   = a - a : by rw sub_self
    ... ≤ b - a : @add_le_add_right R _ a b h (-a)
end

-- ---------------------------------------------------------------------
-- Ejercicio 3. Demostrar que
--    0 ≤ b - a → a ≤ b
-- ---------------------------------------------------------------------

example : 0 ≤ b - a → a ≤ b :=
begin
  intro h,
  calc
    a   = 0 + a       : by rw zero_add
    ... ≤ (b - a) + a : @add_le_add_right R _ 0 (b -a) h a
    ... = b           : by simp
end

-- ---------------------------------------------------------------------
-- Ejercicio 4. Demostrar que
--    a ≤ b
--    0 ≤ c
-- entonces
--    a * c ≤ b * c
-- ---------------------------------------------------------------------

-- 1ª demostración
-- ===============

open_locale classical

example
  (h₁ : a ≤ b)
  (h₂ : 0 ≤ c)
```

```
  : a * c ≤ b * c :=
begin
  by_cases h₃ : b ≤ a,
  { have : a = b,
      { apply le_antisymm h₁ h₃},
    rw this },
  { by_cases h₄ : c = 0,
    { calc a * c = a * 0 : by rw h₄
             ... = 0     : by rw mul_zero
             ... ≤ 0     : by exact le_refl 0
             ... = b * 0 : by rw mul_zero
             ... = b * c : by {congr; rw h₄}},
    { apply le_of_lt,
      apply mul_lt_mul_of_pos_right,
      { exact lt_of_le_not_le h₁ h₃ },
      { exact lt_of_le_of_ne h₂ (ne.symm h₄) }}},
end


-- 2ª demostración
example
  (h₁ : a ≤ b)
  (h₂ : 0 ≤ c)
  : a * c ≤ b * c :=
by exact mul_le_mul_of_nonneg_right h₁ h₂

-- 3ª demostración
example
  (h₁ : a ≤ b)
  (h₂ : 0 ≤ c)
  : a * c ≤ b * c :=
mul_le_mul_of_nonneg_right h₁ h₂
```

2.5.4. Espacios métricos

- Espacios métricos

```
-- ----------------------------------------------------------------------
-- Ejercicio 1. Ejecuta las siguientes acciones
-- 1. Importar la teoría de espacios métricos.
-- 2. Declarar X como un tipo sobre espacios métricos.
-- 3. Declarar x, y y z como variables sobre X.
-- ----------------------------------------------------------------------
```

```
import topology.metric_space.basic       -- 1
variables {X : Type*} [metric_space X]   -- 2
variables x y z : X                      -- 3

-- ---------------------------------------------------------------------
-- Ejercicio 2. Calcular el tipo de las siguientes expresiones
--    dist_self x
--    dist_comm x y
--    dist_triangle x y z
-- ---------------------------------------------------------------------

#check dist_self x
#check dist_comm x y
#check dist_triangle x y z

-- Comentario: Al colocar el cursor sobre check se obtiene
--    dist_self x : dist x x = 0
--    dist_comm x y : dist x y = dist y x
--    dist_triangle x y z : dist x z ≤ dist x y + dist y z
```

- Ejercicio en espacios métricos

```
-- ---------------------------------------------------------------------
-- Ejercicio. Demostrar que en los espacios métricos
--    0 ≤ dist x y
-- ---------------------------------------------------------------------

import topology.metric_space.basic

variables {X : Type*} [metric_space X]

variables x y : X

-- 1ª demostración
example : 0 ≤ dist x y :=
  have 2 * dist x y ≥ 0,
    from calc
      2 * dist x y = dist x y + dist x y : by rw two_mul
               ... = dist x y + dist y x : by rw [dist_comm x y]
               ... ≥ dist x x            : by apply dist_triangle
               ... = 0                   : by rw ← dist_self x,
  nonneg_of_mul_nonneg_left this two_pos
```

```
-- 2ª demostración
example : 0 ≤ dist x y :=
dist_nonneg
```

Capítulo 3

Lógica

En este capítulo se muestra el razonamiento con Lean sobre las conectivas y cuantificadores; es decir, las tácticas para introducirlos en la conclusión o eliminarlos de las hipótesis. Como aplicación, se demostrarán propiedades sobre límites de sucesiones.

3.1. Implicación y cuantificación universal

- Lema con implicaciones y cuantificador universal

```
-- ---------------------------------------------------------------------
-- Ejercicio 1. Importar la librería de los números reales.
-- ---------------------------------------------------------------------

import data.real.basic

-- ---------------------------------------------------------------------
-- Ejercicio 2. Enunciar el lema ej: "para todos los números reales x,
-- y, ε si
--    0 < ε
--    ε ≤ 1
--    abs x < ε
--    abs y < ε
-- entonces
--    abs (x * y) < ε
-- ---------------------------------------------------------------------

lemma ej :
  ∀ x y ε : ℝ,
  0 < ε →
```

```
  ε ≤ 1 →
  abs x < ε →
  abs y < ε →
  abs (x * y) < ε :=
sorry

-- ---------------------------------------------------------------------
-- Ejercicio 3. Crear una sección con las siguientes declaraciones
--    a b δ : ℝ
--    h₀ : 0 < δ
--    h₁ : δ ≤ 1
--    ha : abs a < δ
--    hb : abs b < δ
-- y calcular el tipo de las siguientes expresiones
--    ej a b δ
--    ej a b δ h₀ h₁
--    ej a b δ h₀ h₁ ha hb
-- ---------------------------------------------------------------------

section

variables a b δ : ℝ
variables (h₀ : 0 < δ) (h₁ : δ ≤ 1)
variables (ha : abs a < δ) (hb : abs b < δ)

#check ej a b δ
#check ej a b δ h₀ h₁
#check ej a b δ h₀ h₁ ha hb

-- Comentario: Al colocar el cursor sobre check se obtiene
--    ej a b δ : 0 < δ → δ ≤ 1 → abs a < δ → abs b < δ → abs (a * b) < δ
--    ej a b δ h₀ h₁ : abs a < δ → abs b < δ → abs (a * b) < δ
--    ej a b δ h₀ h₁ ha hb : abs (a * b) < δ

end
```

- Lema con implicaciones y cuantificador universal implícitos

```
-- ---------------------------------------------------------------------
-- Ejercicio 1. Importar la librería de los números reales.
-- ---------------------------------------------------------------------

import data.real.basic
```

```
-- ---------------------------------------------------------------------
-- Ejercicio 2. Enunciar, usando variables implícitas, el lema ej: "para
-- todos los números reales x, y, ε si
--    0 < ε
--    ε ≤ 1
--    abs x < ε
--    abs y < ε
-- entonces
--    abs (x * y) < ε
-- ---------------------------------------------------------------------

lemma ej :
  ∀ {x y ε : ℝ},
  0 < ε →
  ε ≤ 1 →
  abs x < ε →
  abs y < ε →
  abs (x * y) < ε :=
sorry

-- ---------------------------------------------------------------------
-- Ejercicio 3. Crear una sección con las siguientes declaraciones
--    a b δ : ℝ
--    h₀ : 0 < δ
--    h₁ : δ ≤ 1
--    ha : abs a < δ
--    hb : abs b < δ
-- y calcular el tipo de las siguientes expresiones
--    ej h₀ h₁ ha hb
-- ---------------------------------------------------------------------

section

variables a b δ : ℝ
variables (h₀ : 0 < δ) (h₁ : δ ≤ 1)
variables (ha : abs a < δ) (hb : abs b < δ)

#check ej h₀ h₁ ha hb

-- Comentario: Al colocar el cursor sobre check se obtiene
--    ej h₀ h₁ ha hb : abs (a * b) < δ

end
```

- La táctica intros

```
-- ---------------------------------------------------------------------
-- Ejercicio. Demostrar que para todos los números reales x, y, ε si
--    0 < ε
--    ε ≤ 1
--    abs x < ε
--    abs y < ε
-- entonces
--    abs (x * y) < ε
-- ---------------------------------------------------------------------

import data.real.basic tactic

lemma ej :
  ∀ {x y ε : ℝ},
  0 < ε →
  ε ≤ 1 →
  abs x < ε →
  abs y < ε →
  abs (x * y) < ε :=
begin
  intros x y ε epos ele1 xlt ylt,
  by_cases (abs x = 0),
  { calc abs (x * y) = abs x * abs y : by apply abs_mul
                 ... = 0 * abs y     : by rw h
                 ... = 0             : by apply zero_mul
                 ... < ε             : by apply epos
  },
  { have h1 : 0 < abs x,
      { have h2 : 0 ≤ abs x,
          apply abs_nonneg,
        exact lt_of_le_of_ne h2 (ne.symm h)
      },
    calc
      abs (x * y) = abs x * abs y : by rw abs_mul
              ... < abs x * ε     : by apply (mul_lt_mul_left h1).mpr ylt
              ... < ε * ε         : by apply (mul_lt_mul_right epos).mpr xlt
              ... ≤ 1 * ε         : by apply (mul_le_mul_right epos).mpr ele1
              ... = ε             : by rw [one_mul]
  },
end
```

- Definiciones de cotas

```
-- ---------------------------------------------------------------------
-- Ejercicio 1. Importar la librería de los números reales.
-- ---------------------------------------------------------------------

import data.real.basic

-- ---------------------------------------------------------------------
-- Ejercicio 2. Definir la función
--    fn_ub (ℝ → ℝ) → ℝ → Prop
-- tal que (fn_ub f a) afirma que a es una cota superior de f.
-- ---------------------------------------------------------------------

def fn_ub (f : ℝ → ℝ) (a : ℝ) : Prop := ∀ x, f x ≤ a

-- ---------------------------------------------------------------------
-- Ejercicio 3. Definir la función
--    fn_lb (ℝ → ℝ) → ℝ → Prop
-- tal que (fn_lb f a) afirma que a es una cota inferior de f.
-- ---------------------------------------------------------------------

def fn_lb (f : ℝ → ℝ) (a : ℝ) : Prop := ∀ x, a ≤ f x
```

- Suma de cotas superiores

```
-- ---------------------------------------------------------------------
-- Ejercicio 1. Realizar las siguientes acciones:
-- 1. Importar la librería de los números reales.
-- 2. Definir cota superior de una función.
-- 3. Definir cota inferior de una función.
-- 4. Declarar f y g como variables de funciones de ℝ en ℝ.
-- 5. Declarar a y b como variables sobre ℝ.
-- ---------------------------------------------------------------------

import data.real.basic                                  -- 1

def fn_ub (f : ℝ → ℝ) (a : ℝ) : Prop := ∀ x, f x ≤ a    -- 2
def fn_lb (f : ℝ → ℝ) (a : ℝ) : Prop := ∀ x, a ≤ f x    -- 3

variables (f g : ℝ → ℝ)                                 -- 4
variables (a b : ℝ)                                     -- 5

-- ---------------------------------------------------------------------
-- Ejercicio 2. Demostrar que la suma de una cota superior de f y una
-- cota superior de g es una cota superior de f + g.
```

```
-- ----------------------------------------------------------------------

-- 1ª demostración
-- ===============

example
  (hfa : fn_ub f a)
  (hgb : fn_ub g b)
  : fn_ub (λ x, f x + g x) (a + b) :=
begin
  intro x,
  -- dsimp,
  change f x + g x ≤ a + b,
  apply add_le_add,
  apply hfa,
  apply hgb
end

-- Su desarrollo es
--
-- f g : ℝ → ℝ,
-- a b : ℝ,
-- hfa : fn_ub f a,
-- hgb : fn_ub g b
-- ⊢ fn_ub (λ (x : ℝ), f x + g x) (a + b)
--    -- intro x,
-- x : ℝ
-- ⊢ (λ (x : ℝ), f x + g x) x ≤ a + b
--    -- change f x + g x ≤ a + b,
-- ⊢ f x + g x ≤ a + b
--    -- apply add_le_add,
-- | ⊢ f x ≤ a
-- |    -- apply hfa,
-- | ⊢ g x ≤ b
-- |    -- apply hgb
-- no goals

-- Notas.
-- + Nota 1. Con "intro x" se despliega la definición de fn_ub y se introduce
--   la variable x en el contexto.
-- + Nota 2. Con "dsimp" se simplifica la definición del lambda. El mismo
--   efecto se consigue con "change f x + g x ≤ a + b"

-- 2ª demostración
-- ===============
```

```
example
  (hfa : fn_ub f a)
  (hgb : fn_ub g b)
  : fn_ub (λ x, f x + g x) (a + b) :=
λ x, add_le_add (hfa x) (hgb x)
```

- Operaciones con cotas

```
-- ---------------------------------------------------------------------
-- Ejercicio 1. Realizar las siguientes acciones:
-- 1. Importar la librería de los números reales.
-- 2. Definir cota superior de una función.
-- 3. Definir cota inferior de una función.
-- 4. Declarar f y g como variables de funciones de ℝ en ℝ.
-- 5. Declarar a y b como variables sobre ℝ.
-- ---------------------------------------------------------------------

import data.real.basic                                          -- 1

def fn_ub (f : ℝ → ℝ) (a : ℝ) : Prop := ∀ x, f x ≤ a            -- 2
def fn_lb (f : ℝ → ℝ) (a : ℝ) : Prop := ∀ x, a ≤ f x            -- 3

variables (f g : ℝ → ℝ)                                         -- 4
variables (a b : ℝ)                                             -- 5

-- ---------------------------------------------------------------------
-- Ejercicio 2. Demostrar que la suma de una cota inferior de f y una
-- cota inferior de g es una cota inferior de f + g.
-- ---------------------------------------------------------------------

-- 1ª demostración
-- ===============

example
  (hfa : fn_lb f a)
  (hgb : fn_lb g b)
  : fn_lb (λ x, f x + g x) (a + b) :=
begin
  intro x,
  change a + b ≤ f x + g x,
  apply add_le_add,
  apply hfa,
  apply hgb
```

```
end

-- Su desarrollo es
--
-- f g : ℝ → ℝ,
-- a b : ℝ,
-- hfa : fn_lb f a,
-- hgb : fn_lb g b
-- ⊢ fn_lb (λ (x : ℝ), f x + g x) (a + b)
--    -- intro x,
-- x : ℝ
-- ⊢ a + b ≤ (λ (x : ℝ), f x + g x) x
--    -- change a + b ≤ f x + g x,
-- ⊢ a + b ≤ f x + g x
--    -- apply add_le_add,
-- | ⊢ a ≤ f x
-- |    -- apply hfa,
-- | ⊢ b ≤ g x
-- |    -- apply hgb
-- no goals

-- 2ª demostración
-- ===============

example
  (hfa : fn_lb f a)
  (hgb : fn_lb g b)
  : fn_lb (λ x, f x + g x) (a + b) :=
λ x, add_le_add (hfa x) (hgb x)

-- ---------------------------------------------------------------------
-- Ejercicio 3. Demostrar que el producto de dos funciones no negativas
-- es no negativa.
-- ---------------------------------------------------------------------

example
  (nnf : fn_lb f 0)
  (nng : fn_lb g 0)
  : fn_lb (λ x, f x * g x) 0 :=
begin
  intro x,
  change 0 ≤ f x * g x,
  apply mul_nonneg,
  apply nnf,
  apply nng
```

```
end

-- Su desarrollo es
--
-- f g : ℝ → ℝ,
-- nnf : fn_lb f 0,
-- nng : fn_lb g 0
-- ⊢ fn_lb (λ (x : ℝ), f x * g x) 0
--    -- intro x,
-- x : ℝ
-- ⊢ 0 ≤ (λ (x : ℝ), f x * g x) x
--    -- change 0 ≤ f x * g x,
-- ⊢ 0 ≤ f x * g x
--    -- apply mul_nonneg,
-- | ⊢ 0 ≤ f x
-- |    -- apply nnf,
-- | ⊢ 0 ≤ g x
-- |    -- apply nng
-- no goals

-- 2ª demostración
-- ===============

example
  (nnf : fn_lb f 0)
  (nng : fn_lb g 0)
  : fn_lb (λ x, f x * g x) 0 :=
λ x, mul_nonneg (nnf x) (nng x)

-- ---------------------------------------------------------------------
-- Ejercicio 4. Demostrar que si a es una cota superior de f, b es una
-- cota superior de g, a es no negativa y g es no negativa, entonces
-- a * b es una cota superior de f * g.
-- ---------------------------------------------------------------------

example
  (hfa : fn_ub f a)
  (hfb : fn_ub g b)
  (nng : fn_lb g 0)
  (nna : 0 ≤ a)
  : fn_ub (λ x, f x * g x) (a * b) :=
begin
  intro x,
  change f x * g x ≤ a * b,
  apply mul_le_mul,
```

```
  apply hfa,
  apply hfb,
  apply nng,
  apply nna
end

-- Su desarrollo es
--
-- f g : ℝ → ℝ,
-- a b : ℝ,
-- hfa : fn_ub f a,
-- hfb : fn_ub g b,
-- nng : fn_lb g 0,
-- nna : 0 ≤ a
-- ⊢ fn_ub (λ (x : ℝ), f x * g x) (a * b)
--    -- intro x,
-- x : ℝ
-- ⊢ (λ (x : ℝ), f x * g x) x ≤ a * b
--    -- change f x * g x ≤ a * b,
-- ⊢ f x * g x ≤ a * b
--    -- apply mul_le_mul,
-- | ⊢ f x ≤ a
-- |    -- apply hfa,
-- | g x ≤ b
-- |    -- apply hfb,
-- | 0 ≤ g x
-- |    -- apply nng,
-- | 0 ≤ a
-- |    -- apply nna
-- no goals

-- 2ª demostración
-- ===============

example
  (hfa : fn_ub f a)
  (hfb : fn_ub g b)
  (nng : fn_lb g 0)
  (nna : 0 ≤ a)
  : fn_ub (λ x, f x * g x) (a * b) :=
begin
  dunfold fn_ub fn_lb at *,
  intro x,
  have h1:= hfa x,
  have h2:= hfb x,
```

```
  have h3:= nng x,
  exact mul_le_mul h1 h2 h3 nna,
end

-- Prueba
-- ======

/-
f g : ℝ → ℝ,
a b : ℝ,
hfa : fn_ub f a,
hfb : fn_ub g b,
nng : fn_lb g 0,
nna : 0 ≤ a
⊢ fn_ub (λ (x : ℝ), f x * g x) (a * b)
  >> dunfold fn_ub fn_lb at *,
hfa : ∀ (x : ℝ), f x ≤ a,
hfb : ∀ (x : ℝ), g x ≤ b,
nng : ∀ (x : ℝ), 0 ≤ g x,
nna : 0 ≤ a
⊢ ∀ (x : ℝ), f x * g x ≤ a * b
  >> intro x,
x : ℝ
⊢ f x * g x ≤ a * b
  >> have h1:= hfa x,
h1 : f x ≤ a
⊢ f x * g x ≤ a * b
  >> have h2:= hfb x,
h2 : g x ≤ b
⊢ f x * g x ≤ a * b
  >> have h3:= nng x,
h3 : 0 ≤ g x
⊢ f x * g x ≤ a * b
  >> exact mul_le_mul h1 h2 h3 nna,
no goals
-/

-- 3ª demostración
-- ===============

example
  (hfa : fn_ub f a)
  (hfb : fn_ub g b)
  (nng : fn_lb g 0)
  (nna : 0 ≤ a)
```

```
  : fn_ub (λ x, f x * g x) (a * b) :=
begin
  dunfold fn_ub fn_lb at *,
  intro x,
  specialize hfa x,
  specialize hfb x,
  specialize nng x,
  exact mul_le_mul hfa hfb nng nna,
end

-- Prueba
-- ======

/-
f g : ℝ → ℝ,
a b : ℝ,
hfa : fn_ub f a,
hfb : fn_ub g b,
nng : fn_lb g 0,
nna : 0 ≤ a
⊢ fn_ub (λ (x : ℝ), f x * g x) (a * b)
  >> dunfold fn_ub fn_lb at *,
hfa : ∀ (x : ℝ), f x ≤ a,
hfb : ∀ (x : ℝ), g x ≤ b,
nng : ∀ (x : ℝ), 0 ≤ g x,
nna : 0 ≤ a
⊢ ∀ (x : ℝ), f x * g x ≤ a * b
  >> intro x,
x : ℝ
⊢ f x * g x ≤ a * b
  >> specialize hfa x,
hfa : f x ≤ a
⊢ f x * g x ≤ a * b
  >> specialize hfb x,
hfb : g x ≤ b
⊢ f x * g x ≤ a * b
  >> specialize nng x,
nng : 0 ≤ g x
⊢ f x * g x ≤ a * b
  >> exact mul_le_mul hfa hfb nng nna,
no goals
-/

-- 4ª demostración
-- ===============
```

```
example
  (hfa : fn_ub f a)
  (hfb : fn_ub g b)
  (nng : fn_lb g 0)
  (nna : 0 ≤ a)
  : fn_ub (λ x, f x * g x) (a * b) :=
λ x, mul_le_mul (hfa x) (hfb x) (nng x) nna
```

- Cota_doble

```
import import data.real.basic                                    -- 1

-- ---------------------------------------------------------------------
-- Ejercicio 1. Deckarar x como variable implícita sobre los reales.
-- ---------------------------------------------------------------------

variable {x : ℝ}

-- ---------------------------------------------------------------------
-- Ejercicio 2. Demostrar que si
--    ∃a, x < a
-- entonces
--    ∃b, x < b * 2
-- ---------------------------------------------------------------------

-- 1ª demostración
-- ===============

example
  (h : ∃a, x < a)
  : ∃b, x < b * 2 :=
begin
  cases h with a hxa,
  use a / 2,
  calc x   < a         : hxa
       ... = a / 2 * 2 : (div_mul_cancel a two_ne_zero).symm,
end

-- Prueba
-- ======

/-
x : ℝ,
```

```
h : ∃ (a : ℝ), x < a
⊢ ∃ (b : ℝ), x < b * 2
  >> cases h with a hxa,
x a : ℝ,
hxa : x < a
⊢ ∃ (b : ℝ), x < b * 2
  >> use a / 2,
⊢ x < a / 2 * 2
  >> calc x   < a         : hxa
  >>      ... = a / 2 * 2 : (div_mul_cancel a two_ne_zero).symm,
-/

-- Comentario: Se han usado los lemas
-- + div_mul_cancel a : b ≠ 0 → a / b * b = a
-- + two_ne_zero : 2 ≠ 0

-- 2ª demostración
-- ===============

example
  (h : ∃a, x < a)
  : ∃b, x < b * 2 :=
begin
  cases h with a hxa,
  use a / 2,
  linarith,
end

-- Prueba
-- ======

/-
x : ℝ,
h : ∃ (a : ℝ), x < a
⊢ ∃ (b : ℝ), x < b * 2
  >> cases h with a hxa,
hxa : x < a
⊢ ∃ (b : ℝ), x < b * 2
  >> use a / 2,
⊢ x < a / 2 * 2
  >> linarith,
no goals
-/

-- 3ª demostración
```

```
-- ===============

example
  (h : ∃a, x < a)
  : ∃b, x < b * 2 :=
let ⟨a, hxa⟩ := h in ⟨a/2, by linarith⟩
```

- Generalización a monoides

```
-- ---------------------------------------------------------------------
-- Ejercicio 1. Realizar las siguientes acciones:
-- 1. Importar la teoría de grupos ordenados.
-- 2. Declaral α como un tipo.
-- 3. Declarar R como un monoide ordenado cancelativo.
-- 4. Declarar a, b, c y d como variables sobre R.
-- ---------------------------------------------------------------------

import algebra.ordered_group                                    -- 1

variables {α : Type*}                                          -- 2
variables {R : Type*} [ordered_cancel_add_comm_monoid R]       -- 3
variables a b c d : R                                          -- 4

-- ---------------------------------------------------------------------
-- Ejercicio 2. Calcular el tipo de
--     @add_le_add R _ a b c d
-- ---------------------------------------------------------------------

#check @add_le_add R _ a b c d

-- Comentario: Al colocar el cursor sobre check se obtiene
--    a ≤ b → c ≤ d → a + c ≤ b + d

-- ---------------------------------------------------------------------
-- Ejercicio 3. Definir la función
--    fn_ub (α → R) → R → Prop
-- tal que (fn_ub f a) afirma que a es una cota superior de f.
-- ---------------------------------------------------------------------

def fn_ub (f : α → R) (a : R) : Prop := ∀ x, f x ≤ a

-- ---------------------------------------------------------------------
-- Ejercicio 4. Demostrar que que la suma de una cota superior de f y
-- otra de g es una cota superior de f + g.
```

```
-- ----------------------------------------------------------------------

theorem fn_ub_add
  {f g : α → R}
  {a b : R}
  (hfa : fn_ub f a)
  (hgb : fn_ub g b)
  : fn_ub (λ x, f x + g x) (a + b) :=
λ x, add_le_add (hfa x) (hgb x)
```

■ Función monótona

```
-- ----------------------------------------------------------------------
-- Ejercicio. Explicitar la definición de función monótona poniendo el
-- nombre en la hipótesis y su definición en la conclusión.
-- ----------------------------------------------------------------------

import data.real.basic

example
  (f : ℝ → ℝ)
  (h : monotone f) :
  ∀ {a b}, a ≤ b → f a ≤ f b := h
```

■ Suma de funciones monótonas

```
-- ----------------------------------------------------------------------
-- Ejercicio. Demostrar que la suma de dos funciones monótonas es
-- monótona.
-- ----------------------------------------------------------------------

import data.real.basic

variables (f g : ℝ → ℝ)

-- 1ª demostración
-- ===============

example
  (mf : monotone f)
  (mg : monotone g)
  : monotone (λ x, f x + g x) :=
begin
```

```
  intros a b aleb,
  apply add_le_add,
  apply mf aleb,
  apply mg aleb
end

-- Su desarrollo es
--
-- f g : ℝ → ℝ,
-- mf : monotone f,
-- mg : monotone g
-- ⊢ monotone (λ (x : ℝ), f x + g x)
--    >> intros a b aleb,
-- a b : ℝ,
-- aleb : a ≤ b
-- ⊢ (λ (x : ℝ), f x + g x) a ≤ (λ (x : ℝ), f x + g x) b
--    >> apply add_le_add,
-- | ⊢ f a ≤ f b
-- |    >> apply mf aleb,
-- | ⊢ g a ≤ g b
-- |    >> apply mg aleb
-- no goals

-- 2ª demostración
-- ===============

example (mf : monotone f) (mg : monotone g) :
  monotone (λ x, f x + g x) :=
λ a b aleb, add_le_add (mf aleb) (mg aleb)

-- Nota: Se puede iniciar la prueba con
--    λ a b aleb, _
-- situarse en _, pulsar C-c SPC y elegir library_search. Automáticamente, se
-- completa la prueba.
```

- Producto de un positivo por una función monótona

```
-- ---------------------------------------------------------------------
-- Ejercicio. Demostrar que si c es no negativo y f es monótona,
-- entonces c * f es monótona.
-- ---------------------------------------------------------------------

import data.real.basic
```

```
variables (f : ℝ → ℝ)

-- 1ª demostración
-- ===============

example
  {c : ℝ}
  (mf : monotone f)
  (nnc : 0 ≤ c)
  : monotone (λ x, c * f x) :=
begin
  intros a b aleb,
  apply mul_le_mul_of_nonneg_left,
  apply mf aleb,
  apply nnc
end

-- Su desarrollo es
--
-- f : ℝ → ℝ,
-- c : ℝ,
-- mf : monotone f,
-- nnc : 0 ≤ c
-- ⊢ monotone (λ (x : ℝ), c * f x)
--    >> intros a b aleb,
-- a b : ℝ,
-- aleb : a ≤ b
-- ⊢ (λ (x : ℝ), c * f x) a ≤ (λ (x : ℝ), c * f x) b
--    >> apply mul_le_mul_of_nonneg_left,
-- | ⊢ f a ≤ f b
-- |    >> apply mf aleb,
-- | ⊢ 0 ≤ c
-- |    >> apply nnc
-- no goals

-- 2ª demostración
-- ===============

example {c : ℝ} (mf : monotone f) (nnc : 0 ≤ c) :
  monotone (λ x, c * f x) :=
λ a b aleb, mul_le_mul_of_nonneg_left (mf aleb) nnc
```

- Composición de funciones monótonas

```
-- ---------------------------------------------------------------------
-- Ejercicio. Demostrar que la composición de dos funciones monótonas es
-- monótona.
-- ---------------------------------------------------------------------

import data.real.basic

variables (f g : ℝ → ℝ)

-- 1ª demostración
-- ===============

example
  (mf : monotone f)
  (mg : monotone g)
  : monotone (λ x, f (g x)) :=
begin
  intros a b aleb,
  apply mf,
  apply mg,
  apply aleb
end

-- Su desarrollo es
--
-- f g : ℝ → ℝ,
-- mf : monotone f,
-- mg : monotone g
-- ⊢ monotone (λ (x : ℝ), f (g x))
--    >> intros a b aleb,
-- a b : ℝ,
-- aleb : a ≤ b
-- ⊢ (λ (x : ℝ), f (g x)) a ≤ (λ (x : ℝ), f (g x)) b
--    >> apply mf,
-- ⊢ g a ≤ g b
--    >> apply mg,
-- ⊢ a ≤ b
--    >> apply aleb
-- no goals

-- 2ª demostración
-- ===============

example (mf : monotone f) (mg : monotone g) :
  monotone (λ x, f (g x)) :=
```

```
λ a b aleb, mf (mg aleb)
```

- Funciones pares e impares

```
-- ---------------------------------------------------------------------
-- Ejercicio 1. Realizar las siguientes acciones:
-- 1. Importar la teoría de los números reales.
-- 2. Declarar f y g como variables sobre funciones de ℝ en ℝ.
-- ---------------------------------------------------------------------

import data.real.basic                                          -- 1

variables (f g : ℝ → ℝ)                                         -- 2

-- ---------------------------------------------------------------------
-- Ejercicio 2. Definir la función
--    even (ℝ → ℝ) → Prop
-- tal que (even f) afirma que f es par.
-- ---------------------------------------------------------------------

def even (f : ℝ → ℝ) : Prop := ∀ x, f x = f (-x)

-- ---------------------------------------------------------------------
-- Ejercicio 3. Definir la función
--    odd (ℝ → ℝ) → Prop
-- tal que (odd f) afirma que f es impar.
-- ---------------------------------------------------------------------

def odd  (f : ℝ → ℝ) : Prop := ∀ x, f x = - f (-x)

-- ---------------------------------------------------------------------
-- Ejercicio 4. Demostrar que la suma de dos funciones pares es par.
-- ---------------------------------------------------------------------

example
  (ef : even f)
  (eg : even g)
  : even (λ x, f x + g x) :=
begin
  intro x,
  calc
    (λ x, f x + g x) x = f x + g x       : rfl
                   ... = f (-x) + g (-x) : by rw [ef, eg]
end
```

```
-- ---------------------------------------------------------------------
-- Ejercicio 5. Demostrar que la suma de dos funciones impares es par.
-- ---------------------------------------------------------------------

example
  (of : odd f)
  (og : odd g)
  : even (λ x, f x * g x) :=
begin
  intro x,
  calc
    (λ x, f x * g x) x
        = f x * g x               : rfl
    ... = -f (-x) * -g (-x)       : by rw [of, og]
    ... = f (-x) * g (-x)         : by rw neg_mul_neg
    ... = ((λ x, f x * g x) (-x)) : rfl
end

-- ---------------------------------------------------------------------
-- Ejercicio 6. Demostrar que el producto de una función par por una
-- impar es impar.
-- ---------------------------------------------------------------------

example
  (ef : even f)
  (og : odd g)
  : odd (λ x, f x * g x) :=
begin
  intro x,
  calc
    (λ x, f x * g x) x
        = f x * g x               : rfl
    ... = f (-x) * -g (-x)        : by rw [ef, og]
    ... = -(f (-x) * g (-x))      : by rw neg_mul_eq_mul_neg
    ... = -((λ x, f x * g x) (-x)) : rfl
end

-- ---------------------------------------------------------------------
-- Ejercicio 7. Demostrar que si f es par y g es impar, entonces f ∘ g
-- es par.
-- ---------------------------------------------------------------------

example
  (ef : even f)
```

```
  (og : odd g)
  : even (λ x, f (g x)) :=
begin
  intro x,
  calc
    (λ x, f (g x)) x
        = f (g x)               : rfl
    ... = f (-g (-x))           : by rw og
    ... = f (g (-x))            : by rw ← ef
    ... = ((λ x, f (g x)) (-x)) : rfl
end
```

- Propiedad reflexiva del subconjunto

```
-- ---------------------------------------------------------------------
-- Ejercicio. Demostrar que para cualquier conjunto s, s ⊆ s.
-- ---------------------------------------------------------------------

variables {α : Type*} (s : set α)

-- 1ª demostración
-- ===============

example : s ⊆ s :=
by { intros x xs, exact xs }

-- 2ª demostración
-- ===============

example : s ⊆ s :=
λ x xs, xs
```

- Propiedad transitiva del subconjunto

```
-- ---------------------------------------------------------------------
-- Ejercicio. Demostrar la propiedad transitiva de la inclusión de
-- conjuntos.
-- ---------------------------------------------------------------------

import tactic

variables {α : Type*} (r s t : set α)
```

```
-- 1ª demostración
-- ===============

example : r ⊆ s → s ⊆ t → r ⊆ t :=
begin
  intros rs st x xr,
  apply st,
  apply rs,
  exact xr
end

-- El desarrollo es
--
-- α : Type u_1,
-- r s t : set α
-- ⊢ r ⊆ s → s ⊆ t → r ⊆ t
--    >> intros rs st x xr,
-- rs : r ⊆ s,
-- st : s ⊆ t,
-- x : α,
-- xr : x ∈ r
-- ⊢ x ∈ t
--    >> apply st,
-- ⊢ x ∈ s
--    >> apply rs,
-- ⊢ x ∈ r
--    >> exact xr
-- no goals

-- 2ª demostración
-- ===============

example : r ⊆ s → s ⊆ t → r ⊆ t :=
λ rs st x xr, st (rs xr)
```

- Cotas superiores de conjuntos

```
-- ---------------------------------------------------------------------
-- Ejercicio 1. Realizar las siguientes acciones:
-- 1. Declarar α como un tipo sobre órdenes parciales.
-- 2. Declarar s como una variable sobre conjuntos de elementos de tipo α
-- 3. Declarar a y b como variables sobre α.
-- ---------------------------------------------------------------------
```

```
variables {α : Type*} [partial_order α]                                  -- 1
variables (s : set α)                                                    -- 2
variables (a b : α)                                                      -- 3

-- ---------------------------------------------------------------------
-- Ejercicio 2. Definir la función
--    set_ub : set α → α → Prop
-- tal que (set_ub s a) afirma que a es una cota superior de s.
-- ---------------------------------------------------------------------


def set_ub (s : set α) (a : α) := ∀ x, x ∈ s → x ≤ a

-- ---------------------------------------------------------------------
-- Ejercicio 3. Demostrar que si a es una cota superior de s y a ≤ b,
-- entonces b es una cota superior de s.
-- ---------------------------------------------------------------------

example
  (h : set_ub s a)
  (h' : a ≤ b)
  : set_ub s b :=
begin
  intros x xs,
  calc
    x   ≤ a : (h x xs)
    ... ≤ b : h'
end
```

- Funciones inyectivas

```
-- ---------------------------------------------------------------------
-- Ejercicio 1. Realizar las siguientes acciones:
-- 1. Importar la librería de números reales.
-- 2. Abrir el espacio de nombre de las funciones.
-- ---------------------------------------------------------------------

import data.real.basic                                                   -- 1

open function                                                            -- 2

-- ---------------------------------------------------------------------
-- Ejercicio 2. Demostrar que, para todo c la función
--    f(x) = x + c
```

```
-- es inyectiva
-- ---------------------------------------------------------------------

example
  (c : ℝ)
  : injective (λ x, x + c) :=
begin
  intros x₁ x₂ h',
  change x₁ + c = x₂ + c at h',
  apply add_right_cancel h',
end

-- Su desarrollo es
--
-- c : ℝ
-- ⊢ injective (λ (x : ℝ), x + c)
--    >> intros x₁ x₂ h',
-- c x₁ x₂ : ℝ,
-- h' : (λ (x : ℝ), x + c) x₁ = (λ (x : ℝ), x + c) x₂
-- ⊢ x₁ = x₂
--    >> change x₁ + c = x₂ + c at h',
-- c x₁ x₂ : ℝ,
-- h' : x₁ + c = x₂ + c
-- ⊢ x₁ = x₂
--    >> apply add_right_cancel h'
-- no goals

-- 2ª demostración
-- ===============

example
  (c : ℝ)
  : injective (λ x, x + c) :=
begin
  intros x₁ x₂ h',
  apply (add_left_inj c).mp,
  exact h',
end

-- Su desarrollo es
--
-- c : ℝ
-- ⊢ injective (λ (x : ℝ), x + c)
--    >> intros x₁ x₂ h',
-- c x₁ x₂ : ℝ,
```

```
-- h' : (λ (x : ℝ), x + c) x₁ = (λ (x : ℝ), x + c) x₂
-- ⊢ x₁ = x₂
--    >> apply (add_left_inj c).mp,
-- ⊢ x₁ + c = x₂ + c
--    >> exact h',
-- no goals

-- 3ª demostración
-- ===============

example
  (c : ℝ)
  : injective (λ x, x + c) :=
λ x₁ x₂ h', (add_left_inj c).mp h'


-- ----------------------------------------------------------------------
-- Ejercicio 3. Demostrar que, para todo c distinto de cero la función
--    f(x) = x * c
-- es inyectiva
-- ----------------------------------------------------------------------

example
  {c : ℝ}
  (h : c ≠ 0)
  : injective (λ x, c * x) :=
begin
  intros x₁ x₂ h',
  change c * x₁ = c * x₂ at h',
  apply mul_left_cancel' h h',
end

-- Su desarrollo es
--
-- c : ℝ,
-- h : c ≠ 0
-- ⊢ injective (λ (x : ℝ), c * x)
--    >> intros x₁ x₂ h',
-- x₁ x₂ : ℝ,
-- h' : (λ (x : ℝ), c * x) x₁ = (λ (x : ℝ), c * x) x₂
-- ⊢ x₁ = x₂
--    >> change c * x₁ = c * x₂ at h',
-- h' : c * x₁ = c * x₂
-- ⊢ x₁ = x₂
--    >> apply mul_left_cancel' h h',
```

```
-- no goals

-- 2ª demostración
-- ===============

example
  {c : ℝ}
  (h : c ≠ 0)
  : injective (λ x, c * x) :=
begin
  intros x₁ x₂ h',
  apply mul_left_cancel' h,
  exact h',
end

-- Su desarrollo es
--
-- c : ℝ,
-- h : c ≠ 0
-- ⊢ injective (λ (x : ℝ), c * x)
--    >> intros x₁ x₂ h',
-- x₁ x₂ : ℝ,
-- h' : (λ (x : ℝ), c * x) x₁ = (λ (x : ℝ), c * x) x₂
-- ⊢ x₁ = x₂
--    >> apply mul_left_cancel' h,
-- ⊢ c * x₁ = c * x₂
--    >> exact h',
-- no goals

-- 3ª demostración
-- ===============

example
  {c : ℝ}
  (h : c ≠ 0)
  : injective (λ x, c * x) :=
λ x₁ x₂ h', mul_left_cancel' h h'
```

- Composición de funciones inyectivas

```
-- ---------------------------------------------------------------------
-- Ejercicio. Demostrar que la composición de funciones inyectivas es
-- inyectiva.
-- ---------------------------------------------------------------------
```

```
import tactic

open function

variables {α : Type*} {β : Type*} {γ : Type*}
variables {f : α → β} {g : β → γ}

example
  (injg : injective g)
  (injf : injective f) :
  injective (λ x, g (f x)) :=
begin
  intros x₁ x₂ h,
  apply injf,
  apply injg,
  apply h,
end

-- Su desarrollo es
--
-- α : Type u_1,
-- β : Type u_2,
-- γ : Type u_3,
-- f : α → β,
-- g : β → γ,
-- injg : injective g,
-- injf : injective f
-- ⊢ injective (λ (x : α), g (f x))
--    >> intros x₁ x₂ h,
-- x₁ x₂ : α,
-- h : (λ (x : α), g (f x)) x₁ = (λ (x : α), g (f x)) x₂
-- ⊢ x₁ = x₂
--    >> apply injf,
-- ⊢ f x₁ = f x₂
--    >> apply injg,
-- ⊢ g (f x₁) = g (f x₂)
--    >> apply h,
-- no goals
```

3.2. El cuantificador existencial

- Existencia de valor intermedio

```
-- ---------------------------------------------------------------------
-- Ejercicio 1. Demostrar que hay algún número real entre 2 y 3.
-- ---------------------------------------------------------------------

import data.real.basic

-- 1ª demostración
-- ===============

example : ∃ x : ℝ, 2 < x ∧ x < 3 :=
begin
  use 5 / 2,
  norm_num
end

-- Su desarrollo es
--
-- ⊢ ∃ (x : ℝ), 2 < x ∧ x < 3
--    >> use 5 / 2,
-- ⊢ 2 < 5 / 2 ∧ 5 / 2 < 3
--    >> norm_num
-- no goals

-- Comentarios:
-- 1. La táctica (use e) (ver https://bit.ly/3iK14Wk) sustituye la
--    variable del objetivo existencial por la expresión e.
-- 2. La táctica norm_num (ver https://bit.ly/3hoJMgQ) normaliza una
--    expresión numérica.

-- 2ª demostración
-- ===============

example : ∃ x : ℝ, 2 < x ∧ x < 3 :=
begin
  have h : 2 < (5 : ℝ) / 2 ∧ (5 : ℝ) / 2 < 3,
    by norm_num,
  exact ⟨5 / 2, h⟩,
end

-- Su desarrollo es
--
-- ⊢ ∃ (x : ℝ), 2 < x ∧ x < 3
--    >> have h : 2 < (5 : ℝ) / 2 ∧ (5 : ℝ) / 2 < 3,
--    >>   by norm_num,
-- h : 2 < 5 / 2 ∧ 5 / 2 < 3
```

```
-- ⊢ ∃ (x : ℝ), 2 < x ∧ x < 3
--    >> exact ⟨5 / 2, h⟩,
-- no goals

-- Comentario: La táctica (exact ⟨5 / 2, h⟩) sustituye la variable del
-- objetivo por 5/2 y prueba su cuerpo con h.

-- 3ª demostración
-- ===============

example : ∃ x : ℝ, 2 < x ∧ x < 3 :=
⟨5 / 2, by norm_num⟩
```

- Definición de funciones acotadas

```
import data.real.basic

-- ---------------------------------------------------------------------
-- Ejercicio 1. Definir la función
--    fn_ub (ℝ → ℝ) → ℝ → Prop
-- tal que (fn_ub f a) afirma que a es una cota superior de f.
-- ---------------------------------------------------------------------

def fn_ub (f : ℝ → ℝ) (a : ℝ) : Prop := ∀ x, f x ≤ a

-- ---------------------------------------------------------------------
-- Ejercicio 2. Definir la función
--    fn_lb (ℝ → ℝ) → ℝ → Prop
-- tal que (fn_lb f a) afirma que a es una cota inferior de f.
-- ---------------------------------------------------------------------

def fn_lb (f : ℝ → ℝ) (a : ℝ) : Prop := ∀ x, a ≤ f x

-- ---------------------------------------------------------------------
-- Ejercicio 3. Definir la función
--    fn_has_ub (ℝ → ℝ) → Prop
-- tal que (fn_ub f) afirma que f tiene cota superior.
-- ---------------------------------------------------------------------

def fn_has_ub (f : ℝ → ℝ) := ∃ a, fn_ub f a

-- ---------------------------------------------------------------------
-- Ejercicio 4. Definir la función
```

```
--    fn_has_lb (ℝ → ℝ) → Prop
-- tal que (fn_lb f) afirma que f tiene cota inferior.
-- ---------------------------------------------------------------------

def fn_has_lb (f : ℝ → ℝ) := ∃ a, fn_lb f a
```

- Suma de funciones acotadas

```
-- ---------------------------------------------------------------------
-- Ejercicio 1. Realizar las siguientes acciones:
-- 1. Importar la teoría Definicion_de_funciones_acotadas
-- 2. Declarar f y g como variables de funciones de ℝ en ℝ.
-- 3. Declarar a y b como variables sobre ℝ.
-- ---------------------------------------------------------------------

import .Definicion_de_funciones_acotadas -- 1

variables {f g : ℝ → ℝ}                  -- 2
variables {a b : ℝ}                      -- 3

-- ---------------------------------------------------------------------
-- Ejercicio 2. Demostrar que si a es una cota superior de f y b lo es
-- de g, entonces a + b lo es de f + g.
-- ---------------------------------------------------------------------

-- 1ª demostración
-- ===============

example
  (hfa : fn_ub f a)
  (hgb : fn_ub g b)
  : fn_ub (λ x, f x + g x) (a + b) :=
begin
  intro x,
  change f x + g x ≤ a + b,
  apply add_le_add,
  apply hfa,
  apply hgb
end

-- Su desarrollo es
--
-- f g : ℝ → ℝ,
-- a b : ℝ,
```

```
-- hfa : fn_ub f a,
-- hgb : fn_ub g b
-- ⊢ fn_ub (λ (x : ℝ), f x + g x) (a + b)
--    >> intro x,
-- x : ℝ
-- ⊢ (λ (x : ℝ), f x + g x) x ≤ a + b
--    >> change f x + g x ≤ a + b,
-- ⊢ f x + g x ≤ a + b
--    >> apply add_le_add,
-- | ⊢ f x ≤ a
-- |    >> apply hfa,
-- | ⊢ g x ≤ b
-- |    >> apply hgb
-- no goals

theorem fn_ub_add
  (hfa : fn_ub f a)
  (hgb : fn_ub g b)
  : fn_ub (λ x, f x + g x) (a + b) :=
λ x, add_le_add (hfa x) (hgb x)

-- ----------------------------------------------------------------------
-- Ejercicio 3. Demostrar que la suma de dos funciones acotadas
-- superiormente también lo está.
-- ----------------------------------------------------------------------

lemma aux
  (ubf : fn_has_ub f)
  (ubg : fn_has_ub g) :
  fn_has_ub (λ x, f x + g x) :=
begin
  cases ubf with a ubfa,
  cases ubg with b ubfb,
  use a + b,
  apply fn_ub_add ubfa ubfb,
end

-- Su desarrollo es
--
-- f g : ℝ → ℝ,
-- ubf : fn_has_ub f,
-- ubg : fn_has_ub g
-- ⊢ fn_has_ub (λ (x : ℝ), f x + g x)
--    >> cases ubf with a ubfa,
-- f g : ℝ → ℝ,
```

```
-- ubg : fn_has_ub g,
-- a : ℝ,
-- ubfa : fn_ub f a
-- ⊢ fn_has_ub (λ (x : ℝ), f x + g x)
--    >> cases ubg with b ubfb,
-- f g : ℝ → ℝ,
-- a : ℝ,
-- ubfa : fn_ub f a,
-- b : ℝ,
-- ubfb : fn_ub g b
-- ⊢ fn_has_ub (λ (x : ℝ), f x + g x)
--    >> use a + b,
-- ⊢ fn_ub (λ (x : ℝ), f x + g x) (a + b)
--    >> apply fn_ub_add ubfa ubfb
-- no goals
```

- Suma de funciones acotadas inferiormente

```
-- ----------------------------------------------------------------------
-- Ejercicio 1. Realizar las siguientes acciones:
-- 1. Importar la teoría Definicion_de_funciones_acotadas
-- 2. Declarar f y g como variables de funciones de ℝ en ℝ.
-- 3. Declarar a y b como variables sobre ℝ.
-- ----------------------------------------------------------------------

import .Definicion_de_funciones_acotadas -- 1

variables {f g : ℝ → ℝ}                  -- 2
variables {a b : ℝ}                      -- 3

-- ----------------------------------------------------------------------
-- Ejercicio 2. Demostrar que si a es una cota inferior de f y b lo es
-- de g, entonces a + b lo es de f + g.
-- ----------------------------------------------------------------------

-- 1ª demostración
-- ===============

lemma fn_lb_add
  (hfa : fn_lb f a)
  (hgb : fn_lb g b)
  : fn_lb (λ x, f x + g x) (a + b) :=
begin
  intro x,
```

```
  change a + b ≤ f x + g x,
  apply add_le_add,
  apply hfa,
  apply hgb
end

-- Su desarrollo es
--
-- f g : ℝ → ℝ,
-- a b : ℝ,
-- hfa : fn_lb f a,
-- hgb : fn_lb g b
-- ⊢ fn_lb (λ (x : ℝ), f x + g x) (a + b)
--    >> intro x,
-- x : ℝ
-- ⊢ a + b ≤ (λ (x : ℝ), f x + g x) x
--    >> change a + b ≤ f x + g x,
-- ⊢ a + b ≤ f x + g x
--    >> apply add_le_add,
-- | ⊢ a ≤ f x
-- |    >> apply hfa,
-- | ⊢ b ≤ g x
-- |    >> apply hgb
-- no goals

-- 2ª demostración
-- ===============

example
  (hfa : fn_lb f a)
  (hgb : fn_lb g b)
  : fn_lb (λ x, f x + g x) (a + b) :=
λ x, add_le_add (hfa x) (hgb x)

-- ---------------------------------------------------------------------
-- Ejercicio 3. Demostrar que la suma de dos funciones acotadas
-- inferiormente también lo está.
-- ---------------------------------------------------------------------

example
  (lbf : fn_has_lb f)
  (lbg : fn_has_lb g)
  : fn_has_lb (λ x, f x + g x) :=
begin
  cases lbf with a lbfa,
```

```
  cases lbg with b lbgb,
  use a + b,
  apply fn_lb_add lbfa lbgb,
end

-- Su desarrollo es
--
-- f g : ℝ → ℝ,
-- lbf : fn_has_lb f,
-- lbg : fn_has_lb g
-- ⊢ fn_has_lb (λ (x : ℝ), f x + g x)
--    >> cases lbf with a lbfa,
-- f g : ℝ → ℝ,
-- lbg : fn_has_lb g,
-- a : ℝ,
-- lbfa : fn_lb f a
-- ⊢ fn_has_lb (λ (x : ℝ), f x + g x)
--    >> cases lbg with b lbgb,
-- f g : ℝ → ℝ,
-- a : ℝ,
-- lbfa : fn_lb f a,
-- b : ℝ,
-- lbgb : fn_lb g b
-- ⊢ fn_has_lb (λ (x : ℝ), f x + g x)
--    >> use a + b,
-- ⊢ fn_lb (λ (x : ℝ), f x + g x) (a + b)
--    >> apply fn_lb_add lbfa lbgb
-- no goals
```

- Producto por función acotada superiormente

```
-- ----------------------------------------------------------------------
-- Ejercicio 1. Realizar las siguientes acciones:
-- 1. Importar la teoría Definicion_de_funciones_acotadas
-- 2. Declarar f como variable de funciones de ℝ en ℝ.
-- 3. Declarar a y c como variables sobre ℝ.
-- ----------------------------------------------------------------------

import .Definicion_de_funciones_acotadas -- 1

variables {f : ℝ → ℝ}                    -- 2
variables {a c : ℝ}                      -- 3

-- ----------------------------------------------------------------------
```

```
-- Ejercicio 2. Demostrar que si a es una cota superior de f y c no es
-- negativo, entonces c * a es una cota superior de c * f.
-- ----------------------------------------------------------------------

lemma fn_ub_mul
  (hfa : fn_ub f a)
  (h : c ≥ 0)
  : fn_ub (λ x, c * f x) (c * a) :=
λ x, mul_le_mul_of_nonneg_left (hfa x) h

-- ----------------------------------------------------------------------
-- Ejercicio 3. Demostrar que si c ≥ 0 y f está acotada superiormente,
-- entonces c * f también lo está.
-- ----------------------------------------------------------------------

example
  (ubf : fn_has_ub f)
  (h : c ≥ 0)
  : fn_has_ub (λ x, c * f x) :=
begin
  cases ubf with a ubfa,
  use c * a,
  apply fn_ub_mul ubfa h,
end

-- Su desarrollo es
--
-- f : ℝ → ℝ,
-- c : ℝ,
-- ubf : fn_has_ub f,
-- h : c ≥ 0
-- ⊢ fn_has_ub (λ (x : ℝ), c * f x)
--    >> cases ubf with a ubfa,
-- a : ℝ,
-- ubfa : fn_ub f a
-- ⊢ fn_has_ub (λ (x : ℝ), c * f x)
--    >> use c * a,
-- ⊢ fn_ub (λ (x : ℝ), c * f x) (c * a)
--    >> apply fn_ub_mul ubfa h
-- no goals
```

- Sumas de cotas superiores con rcases y rintros

```
-- ----------------------------------------------------------------------
-- Ejercicio 1. Realizar las siguientes acciones:
-- 1. Importar la teoría Definicion_de_funciones_acotadas
-- 2. Declarar f y g como variable de funciones de ℝ en ℝ.
-- 3. Declarar a y c como variables sobre ℝ.
-- ----------------------------------------------------------------------

import .Definicion_de_funciones_acotadas -- 1

variables {f g : ℝ → ℝ}                  -- 2
variables {a b : ℝ}                      -- 3

-- ----------------------------------------------------------------------
-- Ejercicio 2. Demostrar que si a es una cota superior de f y b es una
-- cota superior de g, entonces a + b lo es de f + g.
-- ----------------------------------------------------------------------

theorem fn_ub_add
  (hfa : fn_ub f a)
  (hgb : fn_ub g b)
  : fn_ub (λ x, f x + g x) (a + b) :=
λ x, add_le_add (hfa x) (hgb x)

-- ----------------------------------------------------------------------
-- Ejercicio 3. Demostrar que si f y g está acotadas superiormente,
-- entonces f + g también lo está.
-- ----------------------------------------------------------------------

-- 1ª demostración
-- ===============

example
  (ubf : fn_has_ub f)
  (ubg : fn_has_ub g)
  : fn_has_ub (λ x, f x + g x) :=
begin
  rcases ubf with ⟨a, ubfa⟩,
  rcases ubg with ⟨b, ubfb⟩,
  exact ⟨a + b, fn_ub_add ubfa ubfb⟩,
end

-- Su desarrollo es
--
-- f g : ℝ → ℝ,
-- ubf : fn_has_ub f,
```

```
-- ubg : fn_has_ub g
-- ⊢ fn_has_ub (λ (x : ℝ), f x + g x)
--    >> rcases ubf with ⟨a, ubfa⟩,
-- f g : ℝ → ℝ,
-- ubg : fn_has_ub g,
-- a : ℝ,
-- ubfa : fn_ub f a
-- ⊢ fn_has_ub (λ (x : ℝ), f x + g x)
--    >> rcases ubg with ⟨b, ubfb⟩,
-- f g : ℝ → ℝ,
-- a : ℝ,
-- ubfa : fn_ub f a,
-- b : ℝ,
-- ubfb : fn_ub g b
-- ⊢ fn_has_ub (λ (x : ℝ), f x + g x)
--    >> exact ⟨a + b, fn_ub_add ubfa ubfb⟩
-- no goals

-- 2ª demostración
-- ===============

example :
  fn_has_ub f →
  fn_has_ub g →
  fn_has_ub (λ x, f x + g x) :=
begin
  rintros ⟨a, ubfa⟩ ⟨b, ubfb⟩,
  exact ⟨a + b, fn_ub_add ubfa ubfb⟩,
end

-- Su desarrollo es
--
-- f g : ℝ → ℝ
-- ⊢ fn_has_ub f → fn_has_ub g → fn_has_ub (λ (x : ℝ), f x + g x)
--    >> rintros ⟨a, ubfa⟩ ⟨b, ubfb⟩,
-- f g : ℝ → ℝ,
-- a : ℝ,
-- ubfa : fn_ub f a,
-- b : ℝ,
-- ubfb : fn_ub g b
-- ⊢ fn_has_ub (λ (x : ℝ), f x + g x)
--    >> exact ⟨a + b, fn_ub_add ubfa ubfb⟩
-- no goals

-- 3ª demostración
```

```
-- ===============

example : fn_has_ub f → fn_has_ub g →
  fn_has_ub (λ x, f x + g x) :=
λ ⟨a, ubfa⟩ ⟨b, ubfb⟩, ⟨a + b, fn_ub_add ubfa ubfb⟩
```

- Producto_de_suma_de_cuadrados

```
-- ----------------------------------------------------------------------
-- Ejercicio 1. Realizar las siguientes acciones:
-- 1. Importar la librería de tácticas.
-- 2. Declarar α como un tipo sobre los anillos conmutativos.
-- 3. Declarar x e y como variables sobre α.
-- ----------------------------------------------------------------------

import tactic                           -- 1
variables {α : Type*} [comm_ring α]     -- 2
variables {x y : α}                     -- 3

-- ----------------------------------------------------------------------
-- Ejercicio 2. Definir la función
--    sum_of_squares : α → Prop
-- tal que (sum_of_squares x) afima que x se puede escribir como la suma
-- de dos cuadrados.
-- ----------------------------------------------------------------------

def sum_of_squares (x : α) := ∃ a b, x = a^2 + b^2

-- ----------------------------------------------------------------------
-- Ejercicio 3. Demostrar que si x e y se pueden escribir como la suma
-- de dos cuadrados, entonces también se puede escribir x * y.
-- ----------------------------------------------------------------------

-- 1ª demostración
-- ===============

theorem sum_of_squares_mul
  (sosx : sum_of_squares x)
  (sosy : sum_of_squares y)
  : sum_of_squares (x * y) :=
begin
  rcases sosx with ⟨a, b, xeq⟩,
  rcases sosy with ⟨c, d, yeq⟩,
  rw [xeq, yeq],
```

```
  use [a*c - b*d, a*d + b*c],
  ring,
end

-- Su desarrollo es
--
-- α : Type u_1,
-- _inst_1 : comm_ring α,
-- x y : α,
-- sosx : sum_of_squares x,
-- sosy : sum_of_squares y
-- ⊢ sum_of_squares (x * y)
--    >> rcases sosx with ⟨a, b, xeq⟩,
-- α : Type u_1,
-- _inst_1 : comm_ring α,
-- x y : α,
-- sosy : sum_of_squares y,
-- a b : α,
-- xeq : x = a ^ 2 + b ^ 2
-- ⊢ sum_of_squares (x * y)
--    >> rcases sosy with ⟨c, d, yeq⟩,
-- α : Type u_1,
-- _inst_1 : comm_ring α,
-- x y a b : α,
-- xeq : x = a ^ 2 + b ^ 2,
-- c d : α,
-- yeq : y = c ^ 2 + d ^ 2
-- ⊢ sum_of_squares (x * y)
--    >> rw [xeq, yeq],
-- ⊢ sum_of_squares ((a ^ 2 + b ^ 2) * (c ^ 2 + d ^ 2))
--    >> use [a*c - b*d, a*d + b*c],
-- ⊢ (a^2 + b^2) * (c^2 + d^2) = (a * c - b * d)^2 + (a * d + b * c)^2
--    >> ring
-- no goals

-- 2ª demostración
-- ===============

example
  (sosx : sum_of_squares x)
  (sosy : sum_of_squares y)
  : sum_of_squares (x * y) :=
begin
  rcases sosx with ⟨a, b, rfl⟩,
  rcases sosy with ⟨c, d, rfl⟩,
```

```
  use [a*c - b*d, a*d + b*c],
  ring
end

-- Su desarrollo es
--
-- α : Type u_1,
-- _inst_1 : comm_ring α,
-- x y : α,
-- sosx : sum_of_squares x,
-- sosy : sum_of_squares y
-- ⊢ sum_of_squares (x * y)
--    >> rcases sosx with ⟨a, b, rfl⟩,
-- α : Type u_1,
-- _inst_1 : comm_ring α,
-- x y : α,
-- sosy : sum_of_squares y,
-- a b : α,
-- xeq : x = a ^ 2 + b ^ 2
-- ⊢ sum_of_squares (x * y)
--    >> rcases sosy with ⟨c, d, rfl⟩,
-- α : Type u_1,
-- _inst_1 : comm_ring α,
-- x y a b : α,
-- xeq : x = a ^ 2 + b ^ 2,
-- c d : α,
-- yeq : y = c ^ 2 + d ^ 2
-- ⊢ sum_of_squares (x * y)
--    >> use [a*c - b*d, a*d + b*c],
-- ⊢ (a^2 + b^2) * (c^2 + d^2) = (a * c - b * d)^2 + (a * d + b * c)^2
--    >> ring
-- no goals
```

- Transitividad de la divisibilidad

```
-- ---------------------------------------------------------------------
-- Ejercicio. Demostrar que la relación de divisibilidad es transitiva.
-- ---------------------------------------------------------------------

-- 1ª demostración
-- ===============

import tactic
```

```
variables {a b c : ℕ}

example
  (divab : a ∣ b)
  (divbc : b ∣ c) :
  a ∣ c :=
begin
  cases divab with d beq,
  cases divbc with e ceq,
  rw [ceq, beq],
  use (d * e),
  ring,
end

-- Su desarrollo es
--
-- a b c : ℕ,
-- divab : a ∣ b,
-- divbc : b ∣ c
-- ⊢ a ∣ c
--    >> cases divab with d beq,
-- a b c : ℕ,
-- divbc : b ∣ c,
-- d : ℕ,
-- beq : b = a * d
-- ⊢ a ∣ c
--    >> cases divbc with e ceq,
-- a b c d : ℕ,
-- beq : b = a * d,
-- e : ℕ,
-- ceq : c = b * e
-- ⊢ a ∣ c
--    >> rw [ceq, beq],
-- ⊢ a ∣ a * d * e
--    >> use (d * e),
-- ⊢ a * d * e = a * (d * e)
--    >> ring,
-- no goals

-- 2ª demostración
-- ===============

example
  (divab : a ∣ b)
```

```
  (divbc : b ∣ c) :
  a ∣ c :=
begin
  rcases divbc with ⟨e, rfl⟩,
  rcases divab with ⟨d, rfl⟩,
  use (d * e),
  ring,
end

-- Su desarrollo es
--
-- a b c : ℕ,
-- divab : a ∣ b,
-- divbc : b ∣ c
-- ⊢ a ∣ c
--    >> rcases divbc with ⟨e, rfl⟩,
-- a b : ℕ,
-- divab : a ∣ b,
-- e : ℕ
-- ⊢ a ∣ b * e
--    >> rcases divab with ⟨d, rfl⟩,
-- a e d : ℕ
-- ⊢ a ∣ a * d * e
--    >> use (d * e),
-- ⊢ a * d * e = a * (d * e)
--    >> ring,
-- no goals
```

- Suma divisible

```
-- ----------------------------------------------------------------------
-- Ejercicio 1. Demostrar que si a es un divisor de b y de c, tambien lo
-- es de b + c.
-- ----------------------------------------------------------------------

-- 1ª demostración
-- ===============

import tactic

variables {a b c : ℕ}

example
```

```
  (divab : a ∣ b)
  (divac : a ∣ c) :
  a ∣ (b + c) :=
begin
  cases divab with d beq,
  cases divac with e ceq,
  rw [ceq, beq],
  use (d + e),
  ring,
end

-- Su desarrollo es
--
-- a b c : ℕ,
-- divab : a ∣ b,
-- divac : a ∣ c
-- ⊢ a ∣ b + c
--    >> cases divab with d beq,
-- a b c : ℕ,
-- divac : a ∣ c,
-- d : ℕ,
-- beq : b = a * d
-- ⊢ a ∣ b + c
--    >> cases divac with e ceq,
-- a b c d : ℕ,
-- beq : b = a * d,
-- e : ℕ,
-- ceq : c = a * e
-- ⊢ a ∣ b + c
--    >> rw [ceq, beq],
-- ⊢ a ∣ a * d + a * e
--    >> use (d + e),
-- ⊢ a * d + a * e = a * (d + e)
--    >> ring,
-- no goals

-- 2ª demostración
-- ===============

example
  (divab : a ∣ b)
  (divac : a ∣ c) :
  a ∣ (b + c) :=
begin
```

```
  rcases divab with ⟨d, rfl⟩,
  rcases divac with ⟨e, rfl⟩,
  use (d + e),
  ring,
end

-- Su desarrollo es
--
-- a b c : ℕ,
-- divab : a | b,
-- divac : a | c
-- ⊢ a | b + c
--    >> rcases divab with ⟨d, rfl⟩,
-- a c : ℕ,
-- divac : a | c,
-- d : ℕ
-- ⊢ a | a * d + c
--    >> rcases divac with ⟨e, rfl⟩,
-- ⊢ a | a * d + a * e
--    >> use (d + e),
-- ⊢ a * d + a * e = a * (d + e)
--    >> ring
-- no goals
```

- Suma constante es suprayectiva

```
-- ----------------------------------------------------------------------
-- Ejercicio. Demostrar que para todo número real c, la función
--    f(x) x  + c
-- es suprayectiva.
-- ----------------------------------------------------------------------

import data.real.basic

variable {c : ℝ}

open function

-- 1ª demostración
-- ===============

example : surjective (λ x, x + c) :=
begin
  intro x,
```

```
  use x - c,
  dsimp,
  ring,
end

-- Su desarrollo es
--
-- c : ℝ
-- ⊢ surjective (λ (x : ℝ), x + c)
--    >> intro x,
-- ⊢ ∃ (a : ℝ), (λ (x : ℝ), x + c) a = x
--    >> use x - c,
-- ⊢ (λ (x : ℝ), x + c) (x - c) = x
--    >> dsimp,
-- ⊢ x - c + c = x
--    >> ring,
-- no goals

-- 2ª demostración
-- ===============

example : surjective (λ x, x + c) :=
begin
  intro x,
  use x - c,
  change (x - c) + c = x,
  ring,
end

-- Su desarrollo es
--
-- c : ℝ
-- ⊢ surjective (λ (x : ℝ), x + c)
--    >> intro x,
-- ⊢ ∃ (a : ℝ), (λ (x : ℝ), x + c) a = x
--    >> use x - c,
-- ⊢ (λ (x : ℝ), x + c) (x - c) = x
--    >> change (x - c) + c = x,
-- ⊢ x - c + c = x
--    >> ring,
-- no goals
```

- Producto por no nula es suprayectiva

```
-- ----------------------------------------------------------------------
-- Ejercicio. Demostrar que si c es un número real no nulo, entonces la
-- función
--    f(x) = c * x
-- es suprayectiva.
-- ----------------------------------------------------------------------

import data.real.basic

open function

example
  {c : ℝ}
  (h : c ≠ 0)
  : surjective (λ x, c * x) :=
begin
  intro x,
  use (x / c),
  change c * (x / c) = x,
  rw mul_comm,
  apply div_mul_cancel,
  exact h,
end

-- Su prueba es
--
-- c : ℝ,
-- h : c ≠ 0
-- ⊢ surjective (λ (x : ℝ), c * x)
--    >> intro x,
-- x : ℝ
-- ⊢ ∃ (a : ℝ), (λ (x : ℝ), c * x) a = x
--    >> use (x / c),
-- ⊢ (λ (x : ℝ), c * x) (x / c) = x
--    >> change c * (x / c) = x,
-- ⊢ c * (x / c) = x
--    >> rw mul_comm,
-- ⊢ x / c * c = x
--    >> apply div_mul_cancel,
-- ⊢ c ≠ 0
--    >> exact h,
-- no goals
```

- Propiedad de suprayectivas

```
-- ----------------------------------------------------------------------
-- Ejercicio . Demostrar que si f es una función suprayectiva de ℝ en ℝ,
-- entonces existe un x tal que (f x)^2 = 4.
-- ----------------------------------------------------------------------

import data.real.basic

open function

example
  {f : ℝ → ℝ}
  (h : surjective f)
  : ∃ x, (f x)^2 = 4 :=
begin
  cases h 2 with x hx,
  use x,
  rw hx,
  norm_num,
end

-- La prueba es
--
-- f : ℝ → ℝ,
-- h : surjective f
-- ⊢ ∃ (x : ℝ), f x ^ 2 = 4
--    >> cases h 2 with x hx,
-- x : ℝ,
-- hx : f x = 2
-- ⊢ ∃ (x : ℝ), f x ^ 2 = 4
--    >> use x,
-- ⊢ f x ^ 2 = 4
--    >> rw hx,
-- ⊢ 2 ^ 2 = 4
--    >> norm_num,
-- no goals
```

- Composición de suprayectivas

```
-- ----------------------------------------------------------------------
-- Ejercicio . Demostrar que la composición de funciones suprayectivas
-- es suprayectiva.
-- ----------------------------------------------------------------------


```

```
import tactic

open function

variables {α : Type*} {β : Type*} {γ : Type*}
variables {f : α → β} {g : β → γ}

example
  (surjg : surjective g)
  (surjf : surjective f)
  : surjective (λ x, g (f x)) :=
begin
  intro x,
  cases surjg x with y hy,
  cases surjf y with z hz,
  use z,
  change g (f z) = x,
  rw hz,
  exact hy,
end

-- La prueba es
--
-- α : Type u_1,
-- β : Type u_2,
-- γ : Type u_3,
-- f : α → β,
-- g : β → γ,
-- surjg : surjective g,
-- surjf : surjective f
-- ⊢ surjective (λ (x : α), g (f x))
--    >> intro x,
-- x : γ
-- ⊢ ∃ (a : α), (λ (x : α), g (f x)) a = x
--    >> cases surjg x with y hy,
-- y : β,
-- hy : g y = x
-- ⊢ ∃ (a : α), (λ (x : α), g (f x)) a = x
--    >> cases surjf y with z hz,
-- z : α,
-- hz : f z = y
-- ⊢ ∃ (a : α), (λ (x : α), g (f x)) a = x
--    >> use z,
-- ⊢ (λ (x : α), g (f x)) z = x
--    >> change g (f z) = x,
```

```
-- ⊢ g (f z) = x
--    >> rw hz,
-- ⊢ g y = x
--    >> exact hy
-- no goals
```

3.3. La negación

- Asimétrica implica irreflexiva

```
-- ---------------------------------------------------------------------
-- Ejercicio. Demostrar que para tododos par de numero reales a y b, si
-- a < b entonces no se tiene que b < a.
-- ---------------------------------------------------------------------

import data.real.basic

variables a b : ℝ

example
  (h : a < b)
  : ¬ b < a :=
begin
  intro h',
  have : a < a,
    from lt_trans h h',
  apply lt_irrefl a this,
end

-- La prueba es
--
-- a b : ℝ,
-- h : a < b
-- ⊢ ¬b < a
--    >>   intro h',
-- h' : b < a
-- ⊢ false
--    >>   have : a < a,
--    >>     from lt_trans h h',
-- this : a < a
-- ⊢ false
--    >>   apply lt_irrefl a this,
-- no goals
```

- Función no acotada superiormente

```
-- ---------------------------------------------------------------------
-- Ejercicio. Demostrar que si f es una función de ℝ en ℝ tal que
-- para cada a, existe un x tal que f x > a, entonces f no tiene cota
-- superior.
-- ---------------------------------------------------------------------

import data.real.basic

def fn_ub (f : ℝ → ℝ) (a : ℝ) : Prop := ∀ x, f x ≤ a

def fn_has_ub (f : ℝ → ℝ) := ∃ a, fn_ub f a

variable f : ℝ → ℝ

lemma no_has_ub
  (h : ∀ a, ∃ x, f x > a)
  : ¬ fn_has_ub f :=
begin
  intros fnub,
  cases fnub with a fnuba,
  cases h a with x hx,
  have : f x ≤ a,
    from fnuba x,
  linarith,
end

-- Prueba
-- ------

-- f : ℝ → ℝ,
-- h : ∀ (a : ℝ), ∃ (x : ℝ), f x > a
-- ⊢ ¬fn_has_ub f
--    >> intros fnub,
-- fnub : fn_has_ub f
-- ⊢ false
--    >> cases fnub with a fnuba,
-- a : ℝ,
-- fnuba : fn_ub f a
-- ⊢ false
--    >> cases h a with x hx,
```

```
-- x : ℝ,
-- hx : f x > a
-- ⊢ false
--    >> have : f x ≤ a,
--    >>   from fnuba x,
-- this : f x ≤ a
-- ⊢ false
--    >> linarith,
-- no goals
```

- Función no acotada inferiormente

```
-- ---------------------------------------------------------------------
-- Ejercicio. Demostrar que si f es una función de ℝ en ℝ tal que
-- para cada a, existe un x tal que f x < a, entonces f no tiene cota
-- inferior.
-- ---------------------------------------------------------------------

import data.real.basic

def fn_ub (f : ℝ → ℝ) (a : ℝ) : Prop := ∀ x, f x ≤ a
def fn_lb (f : ℝ → ℝ) (a : ℝ) : Prop := ∀ x, a ≤ f x

def fn_has_ub (f : ℝ → ℝ) := ∃ a, fn_ub f a
def fn_has_lb (f : ℝ → ℝ) := ∃ a, fn_lb f a

variable f : ℝ → ℝ

example
  (h : ∀ a, ∃ x, f x < a)
  : ¬ fn_has_lb f :=
begin
  intros fnlb,
  cases fnlb with a fnlba,
  cases h a with x hx,
  have : a ≤ f x,
    from fnlba x,
  linarith,
end

-- Prueba
-- ------

-- f : ℝ → ℝ,
```

```
-- h : ∀ (a : ℝ), ∃ (x : ℝ), f x < a
-- ⊢ ¬fn_has_lb f
--    >> intros fnlb,
-- fnlb : fn_has_lb f
-- ⊢ false
--    >> cases fnlb with a fnlba,
-- a : ℝ,
-- fnlba : fn_lb f a
-- ⊢ false
--    >> cases h a with x hx,
-- x : ℝ,
-- hx : f x < a
-- ⊢ false
--    >> have : a ≤ f x,
--    >>   from fnlba x,
-- this : a ≤ f x
-- ⊢ false
--    >> linarith,
-- no goals
```

- La identidad no está acotada superiormente

```
-- ---------------------------------------------------------------------
-- Ejercicio. Demostrar que la función identidad no está acotada
-- superiormente.
-- ---------------------------------------------------------------------


import .Funcion_no_acotada_superiormente

example : ¬ fn_has_ub (λ x, x) :=
begin
  apply no_has_ub,
  intro a,
  use a + 1,
  linarith,
end

-- Prueba
-- ------

-- ⊢ ¬fn_has_ub (λ (x : ℝ), x)
--    >> apply no_has_ub,
-- ⊢ ∀ (a : ℝ), ∃ (x : ℝ), x > a
```

```
--    >> intro a,
-- a : ℝ
-- ⊢ ∃ (x : ℝ), x > a
--    >> use a + 1,
-- ⊢ a + 1 > a
--    >> linarith,
-- no goals
```

- Lemas sobre órdenes y negaciones

```
-- ----------------------------------------------------------------------
-- Ejercicio 1. Realizar las siguientes acciones
-- 1. Importar la librería de los reales.
-- 2. Declarar a y b como variables sobre los reales.
-- ----------------------------------------------------------------------

import data.real.basic

variables a b : ℝ

-- ----------------------------------------------------------------------
-- Ejercicio 2. Calcular el tipo de las siguientes expresiones.
--    @not_le_of_gt ℝ _ a b
--    @not_lt_of_ge ℝ _ a b
--    @lt_of_not_ge ℝ _ a b
--    @le_of_not_gt ℝ _ a b
-- ----------------------------------------------------------------------

#check @not_le_of_gt ℝ _ a b
#check @not_lt_of_ge ℝ _ a b
#check @lt_of_not_ge ℝ _ a b
#check @le_of_not_gt ℝ _ a b

-- Comentario: Colocando el cursor sobre check se obtiene
--    not_le_of_gt : a > b → ¬ a ≤ b
--    not_lt_of_ge : a ≥ b → ¬ a < b
--    lt_of_not_ge : ¬ a ≥ b → a < b
--    le_of_not_gt : ¬ a > b → a ≤ b
```

- Propiedades de funciones monótonas

```
-- ----------------------------------------------------------------------
-- Ejercicio 1. Realizar las siguientes acciones
-- 1. Importar la librería de los reales.
-- 2. Declarar f como variable de las funciones de ℝ en ℝ.
-- 3. Declarar a y b como variables sobre los reales.
-- ----------------------------------------------------------------------

import data.real.basic   -- 1
variables (f : ℝ → ℝ)    -- 2
variables (a b : ℝ)      -- 3

-- ----------------------------------------------------------------------
-- Ejercicio 2. Demostrar que si f es monótona y f(a) < f(b), entonces
-- a < b
-- ----------------------------------------------------------------------

example
  (h : monotone f)
  (h' : f a < f b)
  : a < b :=
begin
  apply lt_of_not_ge,
  intro hab,
  have : f a ≥ f b,
    from h hab,
  linarith,
end

-- Prueba
-- ======

-- f : ℝ → ℝ,
-- a b : ℝ,
-- h : monotone f,
-- h' : f a < f b
-- ⊢ a < b
--    >> apply lt_of_not_ge,
-- ⊢ ¬a ≥ b
--    >> intro hab,
-- hab : a ≥ b
-- ⊢ false
--    >> have : f a ≥ f b,
--    >>   from h hab,
-- this : f a ≥ f b
-- ⊢ false
```

```
--    >> linarith,
-- no goals

-- ----------------------------------------------------------------------
-- Ejercicio 3. Demostrar que si
--    a ≤ b
--    f b < f a
-- entonces f no es monótona.
-- ----------------------------------------------------------------------

example
  (h : a ≤ b)
  (h' : f b < f a)
  : ¬ monotone f :=
begin
  intro h1,
  have : f a ≤ f b,
    from h1 h,
  linarith,
end

-- Prueba
-- ======

-- f : ℝ → ℝ,
-- a b : ℝ,
-- h : a ≤ b,
-- h' : f b < f a
-- ⊢ ¬monotone f
--    >> intro h1,
-- h1 : monotone f
-- ⊢ false
--    >> have : f a ≤ f b,
--    >>   from h1 h,
-- this : f a ≤ f b
-- ⊢ false
--    >> linarith,
-- no goals
```

- Condición para no positivo

```
-- ----------------------------------------------------------------------
-- Ejercicio. Sea x un número real tal que para todo número positivo ε,
-- x ≤ ε Demostrar que x ≤ 0.
```

```
-- ----------------------------------------------------------------------


import data.real.basic

-- 1ª demostración
-- ===============

example
  (x : ℝ)
  (h : ∀ ε > 0, x ≤ ε)
  : x ≤ 0 :=
begin
  apply le_of_not_gt,
  intro hx0,
  specialize h (x/2),
  have h1 : x ≤ x / 2,
    { apply h,
      apply half_pos hx0},
  have : x / 2 < x,
    { apply half_lt_self hx0 },
  linarith,
end

-- Prueba
-- ======

-- x : ℝ,
-- h : ∀ (ε : ℝ), ε > 0 → x ≤ ε
-- ⊢ x ≤ 0
--    >> apply le_of_not_gt,
-- ⊢ ¬x > 0
--    >> intro hx0,
-- hx0 : x > 0
-- ⊢ false
--    >> specialize h (x/2),
-- h : x / 2 > 0 → x ≤ x / 2
-- ⊢ false
--    >> have h1 : x ≤ x / 2,
--    >>   { apply h,
-- ⊢ x / 2 > 0
--    >>     apply half_pos hx0},
-- h1 : x ≤ x / 2
-- ⊢ false
--    >> have : x / 2 < x,
```

```
--    >>   { apply half_lt_self hx0 },
-- this : x / 2 < x
-- ⊢ false
--    >> linarith,
-- no goals

-- 2ª demostración
-- ===============

example
  (x : ℝ)
  (h : ∀ ε > 0, x ≤ ε)
  : x ≤ 0 :=
begin
  contrapose! h,
  use x / 2,
  split; linarith,
end

-- Prueba
-- ======

-- x : ℝ,
-- h : ∀ (ε : ℝ), ε > 0 → x ≤ ε
-- ⊢ x ≤ 0
--    >> contrapose! h,
-- h : 0 < x
-- ⊢ ∃ (ε : ℝ), ε > 0 ∧ ε < x
--    >> use x / 2,
-- ⊢ x / 2 > 0 ∧ x / 2 < x
--    >> split; linarith
-- no goals
```

- Negación de cuantificadores

```
-- ----------------------------------------------------------------------
-- Ejercicio 1. Realizar las siguientes acciones:
-- 1. Inportar la librería de tácticas.
-- 2. Declarar α como una variable de tipos.
-- 3. Declarar P una variable sobre las propiedades de α.
-- ----------------------------------------------------------------------

import tactic              -- 1
variables {α : Type*}      -- 2
```

```
variables (P : α → Prop)   -- 3

-- ---------------------------------------------------------------------
-- Ejercicio 2. Demostrar que si
--    ¬ ∃ x, P x
-- entonces
--    ∀ x, ¬ P x
-- ---------------------------------------------------------------------

-- 1ª demostración
-- ===============

example
  (h : ¬ ∃ x, P x)
  : ∀ x, ¬ P x :=
begin
  intros x h1,
  apply h,
  existsi x,
  exact h1,
end

-- Prueba
-- ======

/-
α : Type u_1,
P : α → Prop,
h : ¬∃ (x : α), P x
⊢ ∀ (x : α), ¬P x
  >> intros x h1,
x : α,
h1 : P x
⊢ false
  >> apply h,
⊢ ∃ (x : α), P x
  >> existsi x,
⊢ P x
  >> exact h1,
no goals
-/

-- Comentario: La táctica (existsi e) (ver https://bit.ly/3j21TtU) es la
-- regla de introducción del existencial; es decir, sustituye en el
-- cuerpo del objetivo existencial su variable por e
```

```
example
  (h : ¬ ∃ x, P x)
  : ∀ x, ¬ P x :=
not_exists.mp h

-- 3ª demostración
-- ===============

example
  (h : ¬ ∃ x, P x)
  : ∀ x, ¬ P x :=
by finish

-- 4ª demostración
-- ===============

example
  (h : ¬ ∃ x, P x)
  : ∀ x, ¬ P x :=
by clarify

-- 4ª demostración
-- ===============

example
  (h : ¬ ∃ x, P x)
  : ∀ x, ¬ P x :=
by safe

-- ----------------------------------------------------------------------
-- Ejercicio 3. Demostrar que si
--    ∀ x, ¬ P x
-- entonces
--    ¬ ∃ x, P x
-- ----------------------------------------------------------------------

example
  (h : ∀ x, ¬ P x)
  : ¬ ∃ x, P x :=
begin
  intro h1,
  cases h1 with x hx,
  specialize h x,
  apply h hx,
```

```
end

-- Prueba
-- ======

/-
α : Type u_1,
P : α → Prop,
h : ∀ (x : α), ¬P x
⊢ ¬∃ (x : α), P x
  >> intro h1,
h1 : ∃ (x : α), P x
⊢ false
  >> cases h1 with x hx,
x : α,
hx : P x
⊢ false
  >> specialize h x,
h : ¬P x
⊢ false
  >> apply h hx,
no goals
-/

-- Comentario: La táctica (specialize h e) (ver https://bit.ly/328xYKy)
-- aplica la rela de eliminación del cuantificador universal a la
-- hipótesis h cambiando su variable  por e.

-- 2ª demostración
-- ===============

example
  (h : ∀ x, ¬ P x)
  : ¬ ∃ x, P x :=
not_exists_of_forall_not h

-- 2ª demostración
-- ===============

example
  (h : ∀ x, ¬ P x)
  : ¬ ∃ x, P x :=
by finish

-- ----------------------------------------------------------------------
```

```
-- Ejercicio 4. Habilitar el uso de las reglas de la lógica clásica.
-- ----------------------------------------------------------------------

open_locale classical

-- ----------------------------------------------------------------------
-- Ejercicio 5. Demostrar que si
--    ¬ ∀ x, P x
-- entonces
--    ∃ x, ¬ P x
-- ----------------------------------------------------------------------

-- 1ª demostración
-- ===============

example
  (h : ¬ ∀ x, P x)
  : ∃ x, ¬ P x :=
begin
  by_contradiction h',
  apply h,
  intro x,
  by_contradiction h'',
  exact h' ⟨x, h''⟩,
end

-- Prueba
-- ======

/-
α : Type u_1,
P : α → Prop,
h : ¬∀ (x : α), P x
⊢ ∃ (x : α), ¬P x
  >> by_contradiction h',
h' : ¬∃ (x : α), ¬P x
⊢ false
  >> apply h,
⊢ ∀ (x : α), P x
  >> intro x,
x : α
⊢ P x
  >> by_contradiction h'',
h'' : ¬P x
⊢ false
```

```
  >> exact h' ⟨x, h''⟩,
no goals
-/

-- Comentarios:
-- 1. La táctica (by_contradiction h) es la regla de reducción al
--    absurdo; es decir, si el objetivo es p añade la hipótesis (h : p) y
--    reduce el objetico a false (ver https://bit.ly/2Ckmadb).
-- 2. La táctica (exact h1 ⟨x, h2⟩) es la regla de inntroducción del
--    cuantificador existencial; es decir, si el objetivo es de la forma
--    (∃y, P y) demuestra (P x) con h2 y unifica h1 con (∃x, P x).
--    (ver https://bit.ly/303lLE4).

-- 2ª demostración
-- ===============

example
  (h : ¬ ∀ x, P x)
  : ∃ x, ¬ P x :=
not_forall.mp h

-- 2ª demostración
-- ===============

example
  (h : ¬ ∀ x, P x)
  : ∃ x, ¬ P x :=
by finish

-- ----------------------------------------------------------------------
-- Ejercicio 6. Demostrar que si
--    ∃ x, ¬ P x
-- entonces
--    ¬ ∀ x, P x
-- ----------------------------------------------------------------------

-- 1ª demostración
-- ===============

example
  (h : ∃ x, ¬ P x)
  : ¬ ∀ x, P x :=
begin
  intro h1,
  cases h with x hx,
```

```
  apply hx,
  exact (h1 x),
end

-- Prueba
-- ======

/-
α : Type u_1,
P : α → Prop,
h : ∃ (x : α), ¬P x
⊢ ¬∀ (x : α), P x
  >> intro h1,
h1 : ∀ (x : α), P x
⊢ false
  >> cases h with x hx,
x : α,
hx : ¬P x
⊢ false
  >> apply hx,
⊢ P x
  >> exact (h1 x)
no goals
-/

-- Comentarios:
-- 1. La táctica (intro h), cuando el objetivo es una negación, es la
--    regla de introducción de la negación; es decir, si el objetivo es
--    ¬P entonces añade la hipótesis (h : P) y cambia el objetivo a
--    false.
-- 2. La táctica (cases h with x hx), cuando la hipótesis es un
--    existencial, es la regla de eliminación del existencial; es decir,
--    si h es (∃ (y : α), P y) añade las hipótesis (x : α) y (hx : P x).

-- 2ª demostración
-- ===============

example
  (h : ∃ x, ¬ P x)
  : ¬ ∀ x, P x :=
not_forall.mpr h

-- 3ª demostración
-- ===============
```

```
example
  (h : ∃ x, ¬ P x)
  : ¬ ∀ x, P x :=
by finish
```

- Doble negación

```
-- ----------------------------------------------------------------------
-- Ejercicio 1. Realizar las siguientes acciones:
-- 1. Importar la librería de tácticas.
-- 2. Habilitar la lógica clásica.
-- 3. Declarar Q como una variable proposicional.
-- ----------------------------------------------------------------------

import tactic           -- 1
open_locale classical   -- 2
variable (Q : Prop)     -- 3

-- ----------------------------------------------------------------------
-- Ejercicio 2. Demostrar que si
--     ¬ ¬ Q
-- entonces
--    Q
-- ----------------------------------------------------------------------

-- 1ª demostración
-- ===============

example
  (h : ¬ ¬ Q)
  : Q :=
begin
  by_contradiction h1,
  exact (h h1),
end

-- Prueba
-- ======

/-
Q : Prop,
h : ¬¬Q
⊢ Q
  >> by_contradiction h1,
```

```
h1 : ¬Q
⊢ false
  >> exact (h h1),
no goals
-/

-- 2ª demostración
-- ===============

example
  (h : ¬ ¬ Q)
  : Q :=
not_not.mp h

-- 2ª demostración
-- ===============

example
  (h : ¬ ¬ Q)
  : Q :=
by tauto

-- Comentario: La táctica tauto demuestra las tautologís
-- proposionales.

-- ----------------------------------------------------------------------
-- Ejercicio 3. Demostrar que si
--    Q
-- entonces
--    ¬ ¬ Q
-- ----------------------------------------------------------------------

-- 1ª demostración
-- ===============

example
  (h : Q)
  : ¬ ¬ Q :=
begin
  intro h1,
  exact (h1 h),
end

-- Prueba
-- ======
```

```
/-
Q : Prop,
h : Q
⊢ ¬¬Q
  >> intro h1,
h1 : ¬Q
⊢ false
  >> exact (h1 h)
no goals
-/

-- 2ª demostración
-- ===============

example
  (h : Q)
  : ¬ ¬ Q :=
not_not.mpr h

-- 3ª demostración
-- ===============

example
  (h : Q)
  : ¬ ¬ Q :=
by tauto
```

- CN no acotada superiormente

```
-- ---------------------------------------------------------------------
-- Ejercicio. Sea f una función de ℝ en ℝ. Demostrar que si f no tiene
-- cota superior, entonces para cada a existe un x tal que f(x) > a.
-- ---------------------------------------------------------------------

import data.real.basic

def fn_ub (f : ℝ → ℝ) (a : ℝ) : Prop := ∀ x, f x ≤ a
def fn_has_ub (f : ℝ → ℝ) := ∃ a, fn_ub f a

open_locale classical

variable (f : ℝ → ℝ)
```

```
-- 1ª demostración
-- ===============

example
  (h : ¬ fn_has_ub f)
  : ∀ a, ∃ x, f x > a :=
begin
  intro a,
  by_contradiction h1,
  apply h,
  use a,
  intro x,
  apply le_of_not_gt,
  intro h2,
  apply h1,
  use x,
  exact h2,
end

-- Prueba
-- ======

/-
f : ℝ → ℝ,
h : ¬fn_has_ub f
⊢ ∀ (a : ℝ), ∃ (x : ℝ), f x > a
  >> intro a,
a : ℝ
⊢ ∃ (x : ℝ), f x > a
  >> by_contradiction h1,
h1 : ¬∃ (x : ℝ), f x > a
⊢ false
  >> apply h,
⊢ fn_has_ub f
  >> use a,
⊢ fn_ub f a
  >> intro x,
x : ℝ
⊢ f x ≤ a
  >> apply le_of_not_gt,
⊢ ¬f x > a
  >> intro h2,
h2 : f x > a
⊢ false
  >> apply h1,
```

```
⊢ ∃ (x : ℝ), f x > a
  >> use x,
⊢ f x > a
  >> exact h2,
no goals
-/

-- 2ª demostración
example
  (h : ¬ fn_has_ub f) :
  ∀ a, ∃ x, f x > a :=
begin
  contrapose! h,
  exact h,
end

-- Prueba
-- ======

/-
f : ℝ → ℝ,
h : ¬fn_has_ub f
⊢ ∀ (a : ℝ), ∃ (x : ℝ), f x > a
  >> contrapose! h,
h : ∃ (a : ℝ), ∀ (x : ℝ), f x ≤ a
⊢ fn_has_ub f
  >> exact h,
no goals
-/

-- Comentario: La táctica (contrapose! h) aplica el contrapositivo entre
-- la hipótesis h y el objetivo; es decir, si (h : P) y el objetivo es Q
-- entonces cambia la hipótesis a (h : ¬Q) el objetivo a ¬P aplicando
-- simplificaciones en ambos.
```

- CNS de acotada superiormente (uso de push_neg y simp only)

```
-- ----------------------------------------------------------------------
-- Ejercicio. Realizar las siguientes acciones:
-- 1. Importar la teoría Definicion_de_funciones_acotadas
-- 2. Habilitar la lógica clásica.
-- 3. Declarar f como una variable de ℝ en ℝ.
-- ----------------------------------------------------------------------
```

```
import .Definicion_de_funciones_acotadas   -- 1
open_locale classical                      -- 2
variable (f : ℝ → ℝ)                       -- 3

-- ---------------------------------------------------------------------
-- Ejercicio 2. Demostrar que si
--    ¬ ∀ a, ∃ x, f x > a
-- entonces f está acotada superiormente.
-- ---------------------------------------------------------------------

example
  (h : ¬ ∀ a, ∃ x, f x > a)
  : fn_has_ub f :=
begin
  push_neg at h,
  exact h,
end

-- Prueba
-- ======

/-
f : ℝ → ℝ,
h : ¬∀ (a : ℝ), ∃ (x : ℝ), f x > a
⊢ fn_has_ub f
  >> push_neg at h,
h : ∃ (a : ℝ), ∀ (x : ℝ), f x ≤ a
⊢ fn_has_ub f
  >> exact h,
no goals
-/

-- Comentario. La táctica (push_neg at h) interioriza las negaciones de
-- la hipótesis h.

-- ---------------------------------------------------------------------
-- Ejercicio 3. Demostrar que si f no tiene cota superior, entonces para
-- cada a existe un x tal que f(x) > a.
-- ---------------------------------------------------------------------

example
  (h : ¬ fn_has_ub f)
  : ∀ a, ∃ x, f x > a :=
begin
  simp only [fn_has_ub, fn_ub] at h,
```

```
  push_neg at h,
  exact h,
end

-- Prueba
-- ======

/-
f : ℝ → ℝ,
h : ¬fn_has_ub f
⊢ ∀ (a : ℝ), ∃ (x : ℝ), f x > a
  >> simp only [fn_has_ub, fn_ub] at h,
h : ¬∃ (a : ℝ), ∀ (x : ℝ), f x ≤ a
⊢ ∀ (a : ℝ), ∃ (x : ℝ), f x > a
  >> push_neg at h,
h : ∀ (a : ℝ), ∃ (x : ℝ), a < f x
⊢ ∀ (a : ℝ), ∃ (x : ℝ), f x > a
  >> exact h,
no goals
-/

-- Comentario: La táctica (simp only [h₁, ..., hₙ] at h) simplifica la
-- hipótesis h usando sólo los lemas h₁, ..., hₙ. (Ver
-- https://bit.ly/38O60EV)
```

- CN de no monótona

```
-- ---------------------------------------------------------------------
-- Ejercicio. Demostrar que si f no es monótona, entoces existen x, y
-- tales que x ≤ y y f(y) < f(x).
-- ---------------------------------------------------------------------

import .Definicion_de_funciones_acotadas

open_locale classical

variable (f : ℝ → ℝ)

example
  (h : ¬ monotone f)
  : ∃ x y, x ≤ y ∧ f y < f x :=
begin
  simp only [monotone] at h,
  push_neg at h,
```

```
  exact h,
end

-- Prueba
-- ======

/-
f : ℝ → ℝ,
h : ¬monotone f
⊢ ∃ (x y : ℝ), x ≤ y ∧ f y < f x
  >> simp only [monotone] at h,
h : ¬∀ ⦃a b : ℝ⦄, a ≤ b → f a ≤ f b
⊢ ∃ (x y : ℝ), x ≤ y ∧ f y < f x
  >> push_neg at h,
h : ∃ ⦃a b : ℝ⦄, a ≤ b ∧ f b < f a
⊢ ∃ (x y : ℝ), x ≤ y ∧ f y < f x
  >> exact h,
no goals
-/
```

- Principio de explosión

```
-- ---------------------------------------------------------------------
-- Ejercicio. Demostrar que si 0 < 0, entonces a > 37 para cualquier
-- número a.
-- ---------------------------------------------------------------------

variable a : ℕ

-- 1ª demostración
-- ===============

example
  (h : 0 < 0)
  : a > 37 :=
begin
  exfalso,
  apply lt_irrefl 0 h,
end

-- Prueba
-- ======

/-
```

```
a : ℕ,
h : 0 < 0
⊢ a > 37
  >> exfalso,
a : ℕ,
h : 0 < 0
⊢ false
  >> apply lt_irrefl 0 h,
no goals
-/

-- Comentario: La táctica exfalso sustituye el objetivo por false.

-- 2ª demostración
-- ===============

example
  (h : 0 < 0)
  : a > 37 :=
absurd h (lt_irrefl 0)

-- 3ª demostración
-- ===============

example
  (h : 0 < 0)
  : a > 37 :=
begin
  have h' : ¬ 0 < 0,
    from lt_irrefl 0,
  contradiction,
end

-- Prueba
-- ======

/-
a : ℕ,
h : 0 < 0
⊢ a > 37
  >> have h' : ¬ 0 < 0,
  >>   from lt_irrefl 0,
h' : ¬0 < 0
⊢ a > 37
  >> contradiction,
```

```
no goals
-/

-- Comentario: La táctica contradiction busca dos hipótesis
-- contradictorias.
```

3.4. Conjunción y bicondicional

- Introducción de la conjunción

```
-- ----------------------------------------------------------------------
-- Ejercicio. Sean x e y dos números tales que
--    x ≤ y
--    ¬ y ≤ x
-- entonces
--    x ≤ y ∧ x ≠ y
-- ----------------------------------------------------------------------

import data.real.basic

variables {x y : ℝ}

-- 1ª demostración
-- ===============
example
  (h₀ : x ≤ y)
  (h₁ : ¬ y ≤ x)
  : x ≤ y ∧ x ≠ y :=
begin
  split,
  { assumption },
  intro h,
  apply h₁,
  rw h,
end

-- Prueba
-- ======

/-
x y : ℝ,
h₀ : x ≤ y,
```

```
h₁ : ¬y ≤ x
⊢ x ≤ y ∧ x ≠ y
  >> split,
| ⊢ x ≤ y
|   >> { assumption },
| ⊢ x ≠ y
|   >> intro h,
| h : x = y
| ⊢ false
|   >> apply h₁,
| ⊢ y ≤ x
|   >> rw h.
no goals
-/

-- Comentario: La táctica split, cuando el objetivo es una conjunción
-- (P ∧ Q), aplica la regla de introducción de la conjunción; es decir,
-- sustituye el objetivo por dos nuevos subobjetivos (P y Q).

-- 2ª demostración
-- ===============

example
  (h₀ : x ≤ y)
  (h₁ : ¬ y ≤ x)
  : x ≤ y ∧ x ≠ y :=
⟨h₀, λ h, h₁ (by rw h)⟩

-- Comentario: La notación ⟨h0, h1⟩, cuando el objetivo es una conjunción
-- (P ∧ Q), aplica la regla de introducción de la conjunción donde h0 es
-- una prueba de P y h1 de Q.

-- 3ª demostración
-- ===============

example
  (h₀ : x ≤ y)
  (h₁ : ¬ y ≤ x)
  : x ≤ y ∧ x ≠ y :=
begin
  have h : x ≠ y,
  { contrapose! h₁,
    rw h₁ },
  exact ⟨h₀, h⟩,
end
```

```
-- Prueba
-- ======

/-
x y : ℝ,
h₀ : x ≤ y,
h₁ : ¬y ≤ x
⊢ x ≤ y ∧ x ≠ y
  >> have h : x ≠ y,
  >> { contrapose! h₁,
h₁ : x = y
⊢ y ≤ x
  >>   rw h₁ },
h : x ≠ y
⊢ x ≤ y ∧ x ≠ y
  >> exact ⟨h₀, h⟩,
no goals
-/
```

- Eliminación de la conjunción

```
-- ---------------------------------------------------------------------
-- Ejercicio. Demostrar que en los reales, si
--    x ≤ y ∧ x ≠ y
-- entonces
--    ¬ y ≤ x
-- ---------------------------------------------------------------------

import data.real.basic

variables {x y : ℝ}

-- 1ª demostración
-- ===============

example
  (h : x ≤ y ∧ x ≠ y)
  : ¬ y ≤ x :=
begin
  cases h with h₀ h₁,
  contrapose! h₁,
  exact le_antisymm h₀ h₁,
end
```

```
-- Prueba
-- ======

/-
x y : ℝ,
h : x ≤ y ∧ x ≠ y
⊢ ¬y ≤ x
  >> cases h with h₀ h₁,
h₀ : x ≤ y,
h₁ : x ≠ y
⊢ ¬y ≤ x
  >> contrapose! h₁,
h₀ : x ≤ y,
h₁ : y ≤ x
⊢ x = y
  >> exact le_antisymm h₀ h₁,
no goals
-/


-- Comentario: La táctica (cases h with h₀ h₁,) si la hipótesis h es una
-- conjunción (P ∧ Q), aplica la regla de eliminación de la conjunción;
-- es decir, sustituy h por las hipótesis (h₀ : P) y (h₁ : Q).

-- 2ª demostración
-- ===============

example : x ≤ y ∧ x ≠ y → ¬ y ≤ x :=
begin
  rintros ⟨h₀, h₁⟩ h',
  exact h₁ (le_antisymm h₀ h'),
end

-- Prueba
-- ======

/-
x y : ℝ
⊢ x ≤ y ∧ x ≠ y → ¬y ≤ x
  >> rintros ⟨h₀, h₁⟩ h',
h' : y ≤ x,
h₀ : x ≤ y,
h₁ : x ≠ y
⊢ false
```

```
  >> exact h₁ (le_antisymm h₀ h'),
no goals
-/

-- Comentario: La táctica (rintros ⟨h₀, h₁⟩ h')
-- + si el objetivo es de la forma (P ∧ Q → (R → S)) añade las hipótesis
--   (h₀ : P), (h₁ : Q), (h' : R) y sustituye el objetivo por S.
-- + si el objetivo es de la forma (P ∧ Q → ¬R) añade las hipótesis
--   (h₀ : P), (h₁ : Q), (h' : R) y sustituye el objetivo por false.

-- 3ª demostración
-- ===============

example : x ≤ y ∧ x ≠ y → ¬ y ≤ x :=
λ ⟨h₀, h₁⟩ h', h₁ (le_antisymm h₀ h')

-- 4ª demostración
-- ===============

example
  (h : x ≤ y ∧ x ≠ y)
  : ¬ y ≤ x :=
begin
  intro h',
  apply h.right,
  exact le_antisymm h.left h',
end

-- Prueba
-- ======

/-
x y : ℝ,
h : x ≤ y ∧ x ≠ y
⊢ ¬y ≤ x
  >> intro h',
h' : y ≤ x
⊢ false
  >> apply h.right,
h' : y ≤ x
⊢ x = y
  >> exact le_antisymm h.left h',
no goals
-/
```

```
-- Comentario: Si h es una conjunción (P ∧ Q), entonces h.left es P y
-- h.right es Q.

-- 5ª demostración
-- ===============

example {x y : ℝ} (h : x ≤ y ∧ x ≠ y) : ¬ y ≤ x :=
λ h', h.right (le_antisymm h.left h')
```

▪ Uso de conjunción

```
-- ---------------------------------------------------------------------
-- Ejercicio. Sean m y n números naturales. Demostrar que si
--    m | n ∧ m ≠ n
-- entonces
--    m | n ∧ ¬ n | m
-- ---------------------------------------------------------------------

import data.nat.gcd

open nat

variables {m n : ℕ}

-- 1ª demostración
-- ===============

example
  (h : m ∣ n ∧ m ≠ n)
  : m ∣ n ∧ ¬ n ∣ m :=
begin
  cases h with h₀ h₁,
  split,
    exact h₀,
  contrapose! h₁,
  apply dvd_antisymm h₀ h₁,
end

-- Prueba
-- ======

/-
m n : ℕ,
h : m ∣ n ∧ m ≠ n
```

```
⊢ m ∣ n ∧ ¬n ∣ m
  >> cases h with h₀ h₁,
h₀ : m ∣ n,
h₁ : m ≠ n
⊢ m ∣ n ∧ ¬n ∣ m
  >> split,
| ⊢ m ∣ n
|   >>   exact h₀,
⊢ ¬n ∣ m
  >> contrapose! h₁,
h₁ : n ∣ m
⊢ m = n
  >> apply dvd_antisymm h₀ h₁,
no goals
-/

-- 2ª demostración
-- ===============

example
  (h : m ∣ n ∧ m ≠ n)
  : m ∣ n ∧ ¬ n ∣ m :=
begin
  rcases h with ⟨h₀, h₁⟩,
  split,
    exact h₀,
  contrapose! h₁,
  apply dvd_antisymm h₀ h₁,
end

-- Prueba
-- ======

/-
m n : ℕ,
h : m ∣ n ∧ m ≠ n
⊢ m ∣ n ∧ ¬n ∣ m
  >> rcases h with ⟨h₀, h₁⟩,
h₀ : m ∣ n,
h₁ : m ≠ n
⊢ m ∣ n ∧ ¬n ∣ m
  >> split,
| ⊢ m ∣ n
|   >>   exact h₀,
⊢ ¬n ∣ m
```

```
  >> contrapose! h₁,
h₁ : n ∣ m
⊢ m = n
  >> apply dvd_antisymm h₀ h₁,
no goals
-/
```

- Existenciales y conjunciones anidadas

```
import data.real.basic
import data.nat.gcd

-- 1º ejemplo
-- ==========

-- 1ª demostración
example : ∃ x : ℝ, 2 < x ∧ x < 4 :=
⟨5/2, by norm_num, by norm_num⟩

-- 2ª demostración
example : ∃ x : ℝ, 2 < x ∧ x < 4 :=
begin
  use 5 / 2,
  split; norm_num
end

-- 2º ejemplo
-- ==========

-- 1ª demostración
example (x y : ℝ) : (∃ z : ℝ, x < z ∧ z < y) → x < y :=
begin
  rintros ⟨z, xltz, zlty⟩,
  exact lt_trans xltz zlty
end

-- 2ª demostración
example (x y : ℝ) : (∃ z : ℝ, x < z ∧ z < y) → x < y :=
λ ⟨z, xltz, zlty⟩, lt_trans xltz zlty

-- 3º ejemplo
-- ==========

open nat
```

```
example : ∃ m n : ℕ,
  4 < m ∧ m < n ∧ n < 10 ∧ prime m ∧ prime n :=
begin
  use [5, 7],
  norm_num
end

-- 4º ejemplo
-- ==========

example {x y : ℝ} : x ≤ y ∧ x ≠ y → x ≤ y ∧ ¬ y ≤ x :=
begin
  rintros ⟨h₀, h₁⟩,
  use [h₀, λ h', h₁ (le_antisymm h₀ h')]
end

-- 2ª demostración
example {x y : ℝ} : x ≤ y ∧ x ≠ y → x ≤ y ∧ ¬ y ≤ x :=
begin
  rintros ⟨h₀, h₁⟩,
  split,
    exact h₀,
  contrapose ! h₁,
  apply le_antisymm h₀ h₁
end
```

- Suma nula de dos cuadrados

```
-- ---------------------------------------------------------------------
-- Ejercicio 1. Realizar las siguientes acciones:
-- 1. Importar la teoría de los números reales.
-- 2. Declarar x e y como variables sobre los reales.
-- ---------------------------------------------------------------------

import data.real.basic   -- 1
variables {x y : ℝ}      -- 2

-- ---------------------------------------------------------------------
-- Ejercicio 2. Demostrar que si
--    x^2 + y^2 = 0)
-- entonces
--   x = 0
-- ---------------------------------------------------------------------
```

```
lemma aux
  (h : x^2 + y^2 = 0)
  : x = 0 :=
begin
  have h' : x^2 = 0,
    { apply le_antisymm,
      show x ^ 2 ≤ 0,
        calc
          x ^ 2 ≤ x^2 + y^2 : by simp [le_add_of_nonneg_right,
                                       pow_two_nonneg]
            ... = 0         : by exact h,
      apply pow_two_nonneg },
  exact pow_eq_zero h',
end

-- Prueba
-- ======

/-
x y : ℝ,
h : x ^ 2 + y ^ 2 = 0
⊢ x = 0
  >> have h' : x^2 = 0,
| ⊢ x ^ 2 = 0
|   >>   { apply le_antisymm,
| | ⊢ x ^ 2 ≤ 0
|   >>     show x ^ 2 ≤ 0,
|   >>       calc
|   >>         x ^ 2 ≤ x^2 + y^2 : by simp [le_add_of_nonneg_right,
|   >>                                      pow_two_nonneg]
|   >>           ... = 0         : by exact h,
| ⊢ 0 ≤ x ^ 2
  >>     apply pow_two_nonneg },
h' : x ^ 2 = 0
⊢ x = 0
  >> exact pow_eq_zero h',
no goals
-/

-- Comentario: Los lemas usados son:
-- + le_add_of_nonneg_right : 0 ≤ y → x ≤ x + y
-- + le_antisymm : x ≤ y → y ≤ x → x = y
-- + pow_eq_zero : ∀ {n : ℕ}, x ^ n = 0 → x = 0
-- + pow_two_nonneg x : 0 ≤ x ^ 2
```

```
-- ---------------------------------------------------------------------
-- Ejercicio 3. Demostrar que
--    x^2 + y^2 = 0 ↔ x = 0 ∧ y = 0
-- ---------------------------------------------------------------------

example : x^2 + y^2 = 0 ↔ x = 0 ∧ y = 0 :=
begin
  split,
    intro h,
    split,
     apply (aux h),
    rw add_comm at h,
    apply (aux h),
  intro h1,
  cases h1 with h2 h3,
  rw [h2, h3],
  ring,
end

-- Prueba
-- ======

/-
x y : ℝ
⊢ x ^ 2 + y ^ 2 = 0 ↔ x = 0 ∧ y = 0
  >> split,
| ⊢ x ^ 2 + y ^ 2 = 0 → x = 0 ∧ y = 0
|   >>   intro h,
| h : x ^ 2 + y ^ 2 = 0
| ⊢ x = 0 ∧ y = 0
|   >>   split,
| | ⊢ x = 0
|   >>    apply (aux h),
| h : y ^ 2 + x ^ 2 = 0
| ⊢ y = 0
|   >>   rw add_comm at h,
|   >>   apply (aux h),
⊢ x = 0 ∧ y = 0 → x ^ 2 + y ^ 2 = 0
  >> intro h1,
h1 : x = 0 ∧ y = 0
⊢ x ^ 2 + y ^ 2 = 0
  >> cases h1 with h2 h3,
h2 : x = 0,
h3 : y = 0
```

```
⊢ x ^ 2 + y ^ 2 = 0
  >> rw [h2, h3],
⊢ 0 ^ 2 + 0 ^ 2 = 0
  >> ring,
no goals
-/

-- Comentario: La táctica split, si el objetivo es un bicondicional
-- (P ↔ Q), aplica la introducción del bicondicional; es decir, lo
-- sustituye por dos nuevos objetivos: P → Q y Q → P.
```

- Acotación del valor absoluto

```
-- ---------------------------------------------------------------------
-- Ejercicio. Demostrar eue si
--    |x + 3| < 5
-- entonces
--    -8 < x < 2
-- ---------------------------------------------------------------------

import data.real.basic

-- 1ª demostración
-- ===============

example
  (x y : ℝ)
  : abs (x + 3) < 5 → -8 < x ∧ x < 2 :=
begin
  rw abs_lt,
  intro h,
  split,
   linarith,
  linarith,
end

-- Prueba
-- ======

/-
x y : ℝ
⊢ abs (x + 3) < 5 → -8 < x ∧ x < 2
  >> rw abs_lt,
⊢ -5 < x + 3 ∧ x + 3 < 5 → -8 < x ∧ x < 2
```

```
  >> intro h,
h : -5 < x + 3 ∧ x + 3 < 5
⊢ -8 < x ∧ x < 2
  >> split,
| ⊢ -8 < x
  >>  linarith,
⊢ x < 2
  >> linarith,
no goals
-/

-- Comentario: El lema usado es
-- + abs_lt: abs a < b ↔ -b < a ∧ a < b

-- 2ª demostración
-- ===============

example
  (x y : ℝ)
  : abs (x + 3) < 5 → -8 < x ∧ x < 2 :=
begin
  rw abs_lt,
  intro h,
  split; linarith,
end

-- Comentario: La composición (split; linarith) aplica split y a
-- continuación le aplica linarith a cada subojetivo.
```

- Divisor del mcd

```
-- ---------------------------------------------------------------------
-- Ejercicio. Demostrar que 3 divide al máximo xomún divisor de 6 y 15.
-- ---------------------------------------------------------------------

import data.real.basic
import data.nat.gcd

open nat

-- 1ª demostración
-- ===============

example : 3 | gcd 6 15 :=
```

```
begin
  rw dvd_gcd_iff,
  split; norm_num,
end

-- Prueba
-- ======

/-
⊢ 3 | 6.gcd 15
  rw dvd_gcd_iff,
⊢ 3 | 6 ∧ 3 | 15
  split; norm_num,
no goals
-/

-- Comentario: Se ha usado el lema
-- + dvd_gcd_iff: k | gcd m n ↔ k | m ∧ k | b

-- 2ª demostración
-- ===============

example : 3 ∣ gcd 6 15 :=
dvd_refl 3

-- 3ª demostración
-- ===============

example : 3 ∣ gcd 6 15 :=
by norm_num


-- 4ª demostración
-- ===============

example : 3 ∣ gcd 6 15 :=
by finish

-- 5ª demostración
-- ===============

example : 3 ∣ gcd 6 15 :=
by tauto

-- 6ª demostración
```

```
-- ===============

example : 3 ∣ gcd 6 15 :=
dec_trivial

-- 7ª demostración
-- ===============

example : 3 ∣ gcd 6 15 :=
by refl
```

- Funciones no monótonas

```
-- ---------------------------------------------------------------------
-- Ejercicio 1. Realizar las siguientes acciones:
-- 1. Importar la teoría de los números reales.
-- 2. Declarar f como una variable sobre las funciones de ℝ en ℝ.
-- ---------------------------------------------------------------------

import data.real.basic   -- 1
variable {f : ℝ → ℝ}     -- 2

-- ---------------------------------------------------------------------
-- Ejercicio 2. Demostrar que f es no monótona syss existen x e y tales
-- que x ≤ y y f(x) > f(y).
-- ---------------------------------------------------------------------

-- 1ª demostración
-- ===============

example :
  ¬ monotone f ↔ ∃ x y, x ≤ y ∧ f x > f y :=
begin
  rw [monotone],
  push_neg,
end

-- Prueba
-- ======

/-
f : ℝ → ℝ
⊢ ¬monotone f ↔ ∃ (x y : ℝ), x ≤ y ∧ f x > f y
  >> rw [monotone],
```

```
⊢ (¬∀ ⦃a b : ℝ⦄, a ≤ b → f a ≤ f b) ↔ ∃ (x y : ℝ), x ≤ y ∧ f x > f y
  >> push_neg,
no goals
-/

-- Comentario: Se ha usado la definición
-- + monotone: ∀ ⦃a b : ℝ⦄, a ≤ b → f a ≤ f b

-- 2ª demostración
-- ================

lemma not_monotone_iff :
  ¬ monotone f ↔ ∃ x y, x ≤ y ∧ f x > f y :=
by { rw [monotone], push_neg }

-- ---------------------------------------------------------------------
-- Ejercicio 3. Demostrar que la función opuesta no es monótona.
-- ---------------------------------------------------------------------

example : ¬ monotone (λ x : ℝ, -x) :=
begin
  apply not_monotone_iff.mpr,
  use [2, 3],
  norm_num,
end

-- Prueba
-- ======

/-
⊢ ¬monotone (λ (x : ℝ), -x)
  >> apply not_monotone_iff.mpr,
⊢ ∃ (x y : ℝ), x ≤ y ∧ -x > -y
  >> use [2, 3],
⊢ 2 ≤ 3 ∧ -2 > -3
  >> norm_num,
no goals
-/
```

- Caracterización de menor en órdenes parciales

```
-- ---------------------------------------------------------------------
-- Ejercicio. Demostrar que en un orden parcial
--     a < b ↔ a ≤ b ∧ a ≠ b
```

```
-- ----------------------------------------------------------------------

import tactic

variables {α : Type*} [partial_order α]
variables a b : α

example : a < b ↔ a ≤ b ∧ a ≠ b :=
begin
  rw lt_iff_le_not_le,
  split,
    rintros ⟨h1, h2⟩,
      split,
        exact h1,
        contrapose ! h2,
        rw h2,
    rintros ⟨h3, h4⟩,
      split,
        exact h3,
        contrapose ! h4,
        apply le_antisymm h3 h4,
end

-- Prueba
-- ======

/-α : Type u_1,
_inst_1 : partial_order α,
a b : α
⊢ a < b ↔ a ≤ b ∧ a ≠ b
  >> rw lt_iff_le_not_le,
⊢ a ≤ b ∧ ¬b ≤ a ↔ a ≤ b ∧ a ≠ b
  >> split,
| ⊢ a ≤ b ∧ ¬b ≤ a → a ≤ b ∧ a ≠ b
|   >>   rintros ⟨h1, h2⟩,
| h1 : a ≤ b,
| h2 : ¬b ≤ a
|   >>     split,
| ⊢ a ≤ b ∧ a ≠ b
| | ⊢ a ≤ b
|   >>       exact h1,
| ⊢ a ≠ b
|   >>       contrapose ! h2,
| h2 : a = b
| ⊢ b ≤ a
```

```
|   >>       rw h2,
α : Type u_1,
_inst_1 : partial_order α,
a b : α
⊢ a ≤ b ∧ a ≠ b → a ≤ b ∧ ¬b ≤ a
  >>   rintros ⟨h3, h4⟩,
h3 : a ≤ b,
h4 : a ≠ b
⊢ a ≤ b ∧ ¬b ≤ a
  >>     split,
| ⊢ a ≤ b
|   >>       exact h3,
⊢ ¬b ≤ a
  >>       contrapose ! h4,
h4 : b ≤ a
⊢ a = b
  >>       apply le_antisymm h3 h4,
no goals
-/

-- Comentario: Los lemas usados son
-- + lt_iff_le_not_le : a < b ↔ a ≤ b ∧ ¬b ≤ a
-- + le_antisymm : a ≤ b → b ≤ a → a = b
```

- Irreflexiva y transitiva de menor en preórdenes

```
-- ---------------------------------------------------------------------
-- Ejercicio 1. Realizar las siguientes acciones:
-- 1. Importar la librería de tácticas.
-- 2. Declarar α como una variables sobre preórdenes.
-- 3. Declarar a, b y c como variables sobre elementos de α.
-- ---------------------------------------------------------------------

import tactic                          -- 1
variables {α : Type*} [preorder α]     -- 2
variables a b c : α                    -- 3

-- ---------------------------------------------------------------------
-- Ejercicio 1. Demostrar que que la relación menor es irreflexiva.
-- ---------------------------------------------------------------------

-- 1ª demostración
-- ===============
```

```
example : ¬ a < a :=
begin
  rw lt_iff_le_not_le,
  rintros ⟨h1, h2⟩,
  apply h2 h1,
end

-- Prueba
-- ======

/-
α : Type u_1,
_inst_1 : preorder α,
a : α
⊢ ¬a < a
  >> rw lt_iff_le_not_le,
⊢ ¬(a ≤ a ∧ ¬a ≤ a)
  >> rintros ⟨h1, h2⟩,
h1 : a ≤ a,
h2 : ¬a ≤ a
⊢ false
  >> apply h2 h1,
no goals
-/

-- Comentarios:
-- + La táctica (rintros ⟨h1, h2⟩), si el objetivo es de la forma ¬(P ∧ Q),
--   añade  las hipótesis (h1 : P) y (h2 : Q) y cambia el objetivo a false.
-- + Se ha usado el lema
--      lt_iff_le_not_le : a < b ↔ a ≤ b ∧ ¬b ≤ a

-- 2ª demostración
-- ===============

example : ¬ a < a :=
irrefl a

-- ----------------------------------------------------------------------
-- Ejercicio 3. Demostrar que que la relación menor es transitiva.
-- ----------------------------------------------------------------------

-- 1ª demostración
-- ===============

example : a < b → b < c → a < c :=
```

```
begin
  simp only [lt_iff_le_not_le],
  rintros ⟨h1, h2⟩ ⟨h3, h4⟩,
  split,
    apply le_trans h1 h3,
  contrapose ! h4,
  apply le_trans h4 h1,
end

-- Prueba
-- ======

/-
α : Type u_1,
_inst_1 : preorder α,
a b c : α
⊢ a < b → b < c → a < c
  >> simp only [lt_iff_le_not_le],
⊢ a ≤ b ∧ ¬b ≤ a → b ≤ c ∧ ¬c ≤ b → a ≤ c ∧ ¬c ≤ a
  >> rintros ⟨h1, h2⟩ ⟨h3, h4⟩,
h1 : a ≤ b,
h2 : ¬b ≤ a,
h3 : b ≤ c,
h4 : ¬c ≤ b
⊢ a ≤ c ∧ ¬c ≤ a
  >> split,
| ⊢ a ≤ c
  >>   apply le_trans h1 h3,
⊢ ¬c ≤ a
  >> contrapose ! h4,
h4 : c ≤ a
⊢ c ≤ b
  >> apply le_trans h4 h1,
no goals
-/

-- Comentario: Se ha aplicado los lemas
-- + lt_iff_le_not_le : a < b ↔ a ≤ b ∧ ¬b ≤ a
-- + le_trans : a ≤ b → b ≤ c → a ≤ c

-- 2ª demostración
-- ===============

example : a < b → b < c → a < c :=
lt.trans
```

3.5. Disyunción

- Introducción de la disyunción (Tácticas left / right y lemas or.inl y or.inr)

```
-- ---------------------------------------------------------------------
-- Ejercicio 1. Realizar las siguientes acciones:
-- 1. Importar la librería de los números naturales.
-- 2. Declarar x e y como variables sobre los reales.
-- ---------------------------------------------------------------------

import data.real.basic   -- 1
variables {x y : ℝ}      -- 2

-- ---------------------------------------------------------------------
-- Ejercicio 2. Demostrar que si
--    y > x^2)
-- entonces
--    y > 0 ∨ y < -1
-- ---------------------------------------------------------------------

-- 1ª demostración
-- ===============

example
  (h : y > x^2)
  : y > 0 ∨ y < -1 :=
begin
  left,
  linarith [pow_two_nonneg x],
end

-- Prueba
-- ======

/-
x y : ℝ,
h : y > x ^ 2
⊢ y > 0 ∨ y < -1
  >> left,
⊢ y > 0
  >> linarith [pow_two_nonneg x],
no goals
-/

-- Comentarios:
```

```
-- 1. La táctica left, si el objetivo es una disjunción (P ∨ Q), aplica
--    la regla de introducción de la disyunción; es decir, cambia el
--    objetivo por P. Ver https://bit.ly/3enkT3d
-- 2. Se usa el lema
--       pow_two_nonneg x : 0 ≤ x ^ 2

-- 2ª demostración
-- ===============

example
  (h : y > x^2)
  : y > 0 ∨ y < -1 :=
by { left, linarith [pow_two_nonneg x] }

-- ------------------------------------------------------------------------
-- Ejercicio 3. Demostrar que si
--    -y > x^2 + 1
-- entonces
--    y > 0 ∨ y < -1
-- ------------------------------------------------------------------------

-- 1ª demostración
-- ===============

example
  (h : -y > x^2 + 1)
  : y > 0 ∨ y < -1 :=
begin
  right,
  linarith [pow_two_nonneg x],
end

-- Prueba
-- ======

/-
x y : ℝ,
h : -y > x ^ 2 + 1
⊢ y > 0 ∨ y < -1
  >> right,
⊢ y < -1
  >> linarith [pow_two_nonneg x],
no goals
-/
```

```
-- Comentarios:
-- 1. La táctica right, si el objetivo es una disjunción (P ∨ Q), aplica
--    la regla de introducción de la disyunción; es decir, cambia el
--    objetivo por Q. Ver https://bit.ly/3enkT3d
-- 2. Se usa el lema
--       pow_two_nonneg x : 0 ≤ x ^ 2

-- 2ª demostración
-- ===============

example
  (h : -y > x^2 + 1)
  : y > 0 ∨ y < -1 :=
by { right, linarith [pow_two_nonneg x] }

-- ---------------------------------------------------------------------
-- Ejercicio 4. Demostrar que si
--      y > 0
-- entonces
--      y > 0 ∨ y < -1
-- ---------------------------------------------------------------------

example
  (h : y > 0)
  : y > 0 ∨ y < -1 :=
or.inl h

-- Comentario: Se usa el lema
--    or.inl : a → a ∨ b

-- ---------------------------------------------------------------------
-- Ejercicio 5. Demostrar que si
--    y < -1
-- entonces
--    y > 0 ∨ y < -1
-- ---------------------------------------------------------------------

example
  (h : y < -1)
  : y > 0 ∨ y < -1 :=
or.inr h

-- Comentario: Se usa el lema
--    or.inr : b → a ∨ b
```

- Eliminación de la disyunción (Táctica cases)

```
-- ---------------------------------------------------------------------
-- Ejercicio. Demostrar que para todo par de números reales x e y, si
-- x < |y|, entonces x < y ó x < -y.
-- ---------------------------------------------------------------------

import data.real.basic

variables {x y : ℝ}

-- 1ª Demostración
-- ===============

example : x < abs y → x < y ∨ x < -y :=
begin
  cases le_or_gt 0 y with h1 h2,
  { rw abs_of_nonneg h1,
    intro h3,
    left,
    exact h3 },
  { rw abs_of_neg h2,
    intro h4,
    right,
    exact h4 },
end

-- Prueba
-- ======

/-
x y : ℝ
⊢ x < abs y → x < y ∨ x < -y
  >> cases le_or_gt 0 y with h1 h2,
| h1 : 0 ≤ y
| ⊢ x < abs y → x < y ∨ x < -y
|   >> { rw abs_of_nonneg h1,
| ⊢ x < y → x < y ∨ x < -y
|   >>   intro h3,
| h3 : x < y
| ⊢ x < y ∨ x < -y
|   >>   left,
| ⊢ x < y
|   >>   exact h3 },
```

```
h2 : 0 > y
⊢ x < abs y → x < y ∨ x < -y
  >> { rw abs_of_neg h2,
⊢ x < -y → x < y ∨ x < -y
  >>   intro h4,
h4 : x < -y
⊢ x < y ∨ x < -y
  >>   right,
⊢ x < -y
  >>   exact h4 },
no goals
-/

-- Comentarios:
-- + La táctica (cases h with h1 h2), cuando h es una diyunción, aplica
--   la regla de eliminación de la disyunción; es decir, si h es (P ∨ Q)
--   abre dos casos, en el primero añade la hipótesis (h1 : P) y en el
--   segundo (h2 : Q).
-- + Se han usado los siguientes lemas
--   + le_or_gt x y : x ≤ y ∨ x > y
--   + abs_of_nonneg : 0 ≤ x → abs x = x
--   + abs_of_neg : x < 0 → abs x = -x

-- Comprobación
#check (@le_or_gt ℝ _ x y)
#check (@abs_of_nonneg ℝ _ x)
#check (@abs_of_neg ℝ _ x)

-- 2ª Demostración
-- ===============

example : x < abs y → x < y ∨ x < -y :=
lt_abs.mp

-- Comentario: Se ha usado el lema
-- + lt_abs : x < abs y ↔ x < y ∨ x < -y

-- Comprobación
#check (@lt_abs ℝ _ x y)
```

- Desigualdad triangular para valor absoluto

```
-- ---------------------------------------------------------------------
-- Ejercicio 1. Realizar las siguientes acciones:
-- 1. Importar la librería de los números reales.
-- 2. Declarar x, y, a y b como variables sobre los reales.
-- 3. Crear el espacio de nombres my_abs.
-- ---------------------------------------------------------------------

import data.real.basic    -- 1
variables {x y a b : ℝ}   -- 2
namespace my_abs          -- 3

-- ---------------------------------------------------------------------
-- Ejercicio 2. Demostrar que
--    x ≤ abs x
-- ---------------------------------------------------------------------

-- 1ª demostración
-- ===============

example : x ≤ abs x :=
begin
  cases (le_or_gt 0 x) with h1 h2,
  { rw abs_of_nonneg h1 },
  { rw abs_of_neg h2,
    apply self_le_neg,
    apply le_of_lt h2 },
end

-- Prueba
-- ======

/-
x : ℝ
⊢ x ≤ abs x
  >> cases (le_or_gt 0 x) with h1 h2,
| h1 : 0 ≤ x
| ⊢ x ≤ abs x
  >> { rw abs_of_nonneg h1 },
h2 : 0 > x
⊢ x ≤ abs x
  >> { rw abs_of_neg h2,
⊢ x ≤ -x
  >>   apply self_le_neg,
⊢ x ≤ 0
  >>   apply le_of_lt h2 },
```

```
no goals
-/

-- Comentario: Se han usado los siguientes lemas
-- + le_or_gt x y : x ≤ y ∨ x > y
-- + abs_of_nonneg : 0 ≤ x → abs x = x
-- + abs_of_neg : x < 0 → abs x = -x
-- + self_le_neg : x ≤ 0 → x ≤ -x
-- + le_of_lt : x < y → x ≤ y

-- Comprobación
#check (@le_or_gt ℝ _ x y)
#check (@abs_of_nonneg ℝ _ x)
#check (@abs_of_neg ℝ _ x)
#check (@self_le_neg ℝ _ x)
#check (@le_of_lt ℝ _ x y)

-- 2ª demostración
-- ===============

example : x ≤ abs x :=
begin
  unfold abs,
  exact le_max_left x (-x),
end

-- Comentarios:
-- 1. La táctica (unfold e) despliega la definición de e.
-- 2. La definición de abs
--    + abs (a : α) : α := max a (-a)
-- 3. Se ha usado el lema
--    + le_max_left x y : x ≤ max x y

-- Comprobación
#check (@le_max_left ℝ _ x y)
#print abs

-- 3ª demostración
-- ===============

example : x ≤ abs x :=
le_max_left x (-x)

-- ---------------------------------------------------------------------
-- Ejercicio 3. Demostrar que
```

```
--    -x ≤ abs x
-- ---------------------------------------------------------------------

-- 1ª demostración
-- ===============

theorem neg_le_abs_self : -x ≤ abs x :=
begin
  cases (le_or_gt 0 x) with h1 h2,
  { rw abs_of_nonneg h1,
    apply neg_le_self h1 },
  { rw abs_of_neg h2 },
end

-- Prueba
-- ======

/-
x : ℝ
⊢ -x ≤ abs x
  >> cases (le_or_gt 0 x) with h1 h2,
| h1 : 0 ≤ x
| ⊢ -x ≤ abs x
|   >> { rw abs_of_nonneg h1,
| ⊢ -x ≤ x
|   >>   apply neg_le_self h1 },
h2 : 0 > x
⊢ -x ≤ abs x
  >> { rw abs_of_neg h2 },
no goals
-/

-- Comentario: Los lemas utilizados son
-- + le_or_gt x y : x ≤ y ∨ x > y
-- + abs_of_nonneg : 0 ≤ x → abs x = x
-- + neg_le_self : 0 ≤ x → -x ≤ x
-- + abs_of_neg : x < 0 → abs x = -x

-- Comprobación
#check (@le_or_gt ℝ _ x y)
#check (@abs_of_nonneg ℝ _ x)
#check (@neg_le_self ℝ _ x)
#check (@abs_of_neg ℝ _ x)

-- 2ª demostración
```

```
-- ===============

example : -x ≤ abs x :=
begin
  unfold abs,
  exact le_max_right x (-x),
end

-- Comentarios:
-- 1. La táctica (unfold e) despliega la definición de e.
-- 2. La definición de abs
--    + abs (a : α) : α := max a (-a)
-- 3. Se ha usado el lema
--    + le_max_right x y : y ≤ max x y

-- Comprobación:
#check (@le_max_right ℝ _ x y)

-- 3ª demostración
-- ===============

example : -x ≤ abs x :=
le_max_right x (-x)

-- ---------------------------------------------------------------------
-- Ejercicio 4. Demostrar que si
--    0 ≤ a + b
--    0 ≤ a
-- entonces
--    abs (a + b) ≤ abs a + abs b
-- ---------------------------------------------------------------------

open_locale classical

lemma aux1
  (h1 : 0 ≤ a + b)
  (h2 : 0 ≤ a)
  : abs (a + b) ≤ abs a + abs b :=
begin
  by_cases h3 : 0 ≤ b,
  show abs (a + b) ≤ abs a + abs b,
    calc
      abs (a + b) ≤ abs (a + b)   : by apply le_refl
              ... = a + b         : by rw (abs_of_nonneg h1)
              ... = abs a + b     : by rw (abs_of_nonneg h2)
```

```
              ... = abs a + abs b : by rw (abs_of_nonneg h3),
  have h4 : b ≤ 0,
    from le_of_lt (lt_of_not_ge h3),
  show abs (a + b) ≤ abs a + abs b,
    calc
      abs (a + b) = a + b         : by rw (abs_of_nonneg h1)
              ... = abs a + b     : by rw (abs_of_nonneg h2)
              ... ≤ abs a + 0     : add_le_add_left h4 _
              ... ≤ abs a + -b    : add_le_add_left (neg_nonneg_of_nonpos h4) _
              ... = abs a + abs b : by rw (abs_of_nonpos h4),
end

-- Prueba
-- ======

/-
a b : ℝ,
h1 : 0 ≤ a + b,
h2 : 0 ≤ a
⊢ abs (a + b) ≤ abs a + abs b
  >> by_cases h3 : 0 ≤ b,
| h3 : 0 ≤ b
| ⊢ abs (a + b) ≤ abs a + abs b
|   >> show abs (a + b) ≤ abs a + abs b, calc
|   >>   abs (a + b) ≤ abs (a + b)   : by apply le_refl
|   >>           ... = a + b         : by rw (abs_of_nonneg h1)
|   >>           ... = abs a + b     : by rw (abs_of_nonneg h2)
|   >>           ... = abs a + abs b : by rw (abs_of_nonneg h3),
h3 : ¬0 ≤ b
⊢ abs (a + b) ≤ abs a + abs b
  >> have h4 : b ≤ 0,
  >>   from le_of_lt (lt_of_not_ge h3),
h4 : b ≤ 0
⊢ abs (a + b) ≤ abs a + abs b
  >> show abs (a + b) ≤ abs a + abs b, calc
  >>   abs (a + b) = a + b         : by rw (abs_of_nonneg h1)
  >>           ... = abs a + b     : by rw (abs_of_nonneg h2)
  >>           ... ≤ abs a + 0     : add_le_add_left h4 _
  >>           ... ≤ abs a + -b    : add_le_add_left (neg_nonneg_of_nonpos h4)
  >>           ... = abs a + abs b : by rw (abs_of_nonpos h4),
-/

-- Comentarios:
-- 1. La táctica (by_cases h : p) aplica el principio de tercio excluso
--    sobre p; es decir, considera dos casos: en el primero le añade la
```

```
--    hipótesis (h : p) y en el segundo, (h : ¬p).
-- 2. Para aplicar la táctica by_cases hay que habilitar la lógica
--    clásica.
-- 3. Se han usado los siguientes lemas
--    + le_refl x : x ≤ x
--    + abs_of_nonneg : 0 ≤ x → abs x = x
--    + le_of_lt : x < y → x ≤ y
--    + lt_of_not_ge : ¬x ≥ y → x < y
--    + add_le_add_left : x ≤ y → ∀ (c : ℝ), c + x ≤ c + y
--    + neg_nonneg_of_nonpos : x ≤ 0 → 0 ≤ -x
--    + abs_of_nonpos : x ≤ 0 → abs x = -x

-- Comprobación:
#check (@le_refl ℝ _ x)
#check (@abs_of_nonneg ℝ _ x)
#check (@le_of_lt ℝ _ x y)
#check (@lt_of_not_ge ℝ _ x y)
#check (@add_le_add_left ℝ _ x y)
#check (@neg_nonneg_of_nonpos ℝ _ x)
#check (@abs_of_nonpos ℝ _ x)

-- ---------------------------------------------------------------------
-- Ejercicio 5. Demostrar que si
--    0 ≤ a + b
-- entonces
--    abs (a + b) ≤ abs a + abs b
-- ---------------------------------------------------------------------

lemma aux2
  (h1 : 0 ≤ a + b)
  : abs (a + b) ≤ abs a + abs b :=
begin
  by_cases h2 : 0 ≤ a,
    exact @aux1 a b h1 h2,
  rw add_comm at h1,
  have h3 : 0 ≤ b,
    linarith,
  rw add_comm,
  rw add_comm (abs a),
  exact @aux1 b a h1 h3,
end

-- ---------------------------------------------------------------------
-- Ejercicio 6. Demostrar que
--    abs (x + y) ≤ abs x + abs y
```

```
-- ----------------------------------------------------------------------

theorem abs_add : abs (x + y) ≤ abs x + abs y :=
begin
  by_cases h2 : 0 ≤ x + y,
    { exact @aux2 x y h2 },
  have h3 : x + y ≤ 0,
    by exact le_of_not_ge h2,
  have h4: -x + -y = -(x + y),
    by rw neg_add,
  have h5 : 0 ≤ -(x + y),
    from neg_nonneg_of_nonpos h3,
  have h6 : 0 ≤ -x + -y,
    { rw [← h4] at h5,
      exact h5 },
  calc
     abs (x + y) = abs (-x + -y)        : by rw [← abs_neg, neg_add]
             ... ≤ abs (-x) + abs (-y) : aux2 h6
             ... = abs x + abs y        : by rw [abs_neg, abs_neg],
end

-- Prueba
-- ======

/-
x y : ℝ
⊢ abs (x + y) ≤ abs x + abs y
  >> by_cases h2 : 0 ≤ x + y,
| h2 : 0 ≤ x + y
| ⊢ abs (x + y) ≤ abs x + abs y
|   >>   { exact @aux2 x y h2 },
h2 : ¬0 ≤ x + y
⊢ abs (x + y) ≤ abs x + abs y
  >> have h3 : x + y ≤ 0,
  >>   by exact le_of_not_ge h2,
x y : ℝ,
h2 : ¬0 ≤ x + y,
h3 : x + y ≤ 0
⊢ abs (x + y) ≤ abs x + abs y
  >> have h4: -x + -y = -(x + y),
  >>   by rw neg_add,
x y : ℝ,
h2 : ¬0 ≤ x + y,
h3 : x + y ≤ 0,
h4 : -x + -y = -(x + y)
```

```
⊢ abs (x + y) ≤ abs x + abs y
  >> have h5 : 0 ≤ -(x + y),
  >>   from neg_nonneg_of_nonpos h3,
x y : ℝ,
h2 : ¬0 ≤ x + y,
h3 : x + y ≤ 0,
h4 : -x + -y = -(x + y),
h5 : 0 ≤ -(x + y)
⊢ abs (x + y) ≤ abs x + abs y
  >> have h6 : 0 ≤ -x + -y,
  >>   { rw [← h4] at h5,
x y : ℝ,
h2 : ¬0 ≤ x + y,
h3 : x + y ≤ 0,
h4 : -x + -y = -(x + y),
h5 : 0 ≤ -x + -y
⊢ 0 ≤ -x + -y
  >>     exact h5 },
x y : ℝ,
h2 : ¬0 ≤ x + y,
h3 : x + y ≤ 0,
h4 : -x + -y = -(x + y),
h5 : 0 ≤ -(x + y),
h6 : 0 ≤ -x + -y
⊢ abs (x + y) ≤ abs x + abs y
  >> calc
  >>    abs (x + y) = abs (-x + -y)       : by rw [← abs_neg, neg_add]
  >>             ... ≤ abs (-x) + abs (-y) : aux2 h6
  >>             ... = abs x + abs y       : by rw [abs_neg, abs_neg],
-/

-- Comentario: Se han usado los lemas
-- + le_of_not_ge : ¬x ≥ y → x ≤ y
-- + neg_add x y : -(x + y) = -x + -y
-- + neg_nonneg_of_nonpos : x ≤ 0 → 0 ≤ -x

-- Comprobación:
#check (@le_of_not_ge ℝ _ x y)
#check (@neg_add ℝ _ x y)
#check (@neg_nonneg_of_nonpos ℝ _ x)

end my_abs
```

- Cotas del valor absoluto

```
-- ----------------------------------------------------------------------
-- Ejercicio 1. Realizar las siguientes acciones:
-- 1. Importar la teoría de los números reales.
-- 2. Declarar x e y como variables sobre los reales.
-- 3. Iniciar el espacio de nombre my_abs.
-- ----------------------------------------------------------------------

import data.real.basic   -- 1
variables {x y z : ℝ}    -- 2
namespace my_abs         -- 3

-- ----------------------------------------------------------------------
-- Ejercicio 2. Demostrar que
--    x < abs y ↔ x < y ∨ x < -y
-- ----------------------------------------------------------------------

-- 1ª demostración
-- ===============

theorem lt_abs : x < abs y ↔ x < y ∨ x < -y :=
begin
  unfold abs,
  exact lt_max_iff,
end

-- Prueba
-- ======

/-
x y : ℝ
⊢ x < abs y ↔ x < y ∨ x < -y
  >> unfold abs,
⊢ x < max y (-y) ↔ x < y ∨ x < -y
  >> exact lt_max_iff,
no goals
-/

-- Comentarios:
-- 1. La táctica (unfold e) despliega la definición de e.
-- 2. La definición de abs
--    + abs (a : α) : α := max a (-a)
-- 3. Se ha usado el lema
--    + lt_max_iff : x < max y z ↔ x < y ∨ x < z

-- Comprobación
```

```
#check (@lt_max_iff ℝ _ x y z)

-- 2ª demostración
-- ===============

example : x < abs y ↔ x < y ∨ x < -y :=
lt_max_iff

-- ---------------------------------------------------------------------
-- Ejercicio 2. Demostrar que
--    abs x < y ↔ - y < x ∧ x < y
-- ---------------------------------------------------------------------

theorem abs_lt : abs x < y ↔ -y < x ∧ x < y :=
begin
  unfold abs,
  split,
    { intro h1,
      rw max_lt_iff at h1,
      cases h1 with h2 h3,
      split,
        { exact neg_lt.mp h3 },
        { exact h2 }},
    { intro h4,
      apply max_lt_iff.mpr,
      cases h4 with h5 h6,
      split,
       { exact h6 },
       { exact neg_lt.mp h5 }},
end

-- Prueba
-- ======

/-
x y : ℝ
⊢ abs x < y ↔ -y < x ∧ x < y
  unfold abs,
⊢ max x (-x) < y ↔ -y < x ∧ x < y
  >> split,
| ⊢ max x (-x) < y → -y < x ∧ x < y
|   >>   { intro h1,
| h1 : max x (-x) < y
| ⊢ -y < x ∧ x < y
|   >>     rw max_lt_iff at h1,
```

```
| h1 : x < y ∧ -x < y
| ⊢ -y < x ∧ x < y
|   >>     cases h1 with h2 h3,
| h2 : x < y,
| h3 : -x < y
| ⊢ -y < x ∧ x < y
|   >>     split,
| | ⊢ -y < x
|   >>       { exact neg_lt.mp h3 },
| ⊢ x < y
|   >>       { exact h2 }},
⊢ -y < x ∧ x < y → max x (-x) < y
  >>   { intro h4,
h4 : -y < x ∧ x < y
⊢ max x (-x) < y
  >>     apply max_lt_iff.mpr,
⊢ x < y ∧ -x < y
  >>     cases h4 with h5 h6,
h5 : -y < x,
h6 : x < y
⊢ x < y ∧ -x < y
  >>     split,
| ⊢ x < y
  >>      { exact h6 },
⊢ -x < y
  >>      { exact neg_lt.mp h5 }},
no goals
-/

-- Comentarios: Se han usado los siguientes lemas:
-- + max_lt_iff : max x y < z ↔ x < z ∧ y < z
-- + neg_lt : -x < y ↔ -y < x

-- Comprobación:
#check (@max_lt_iff ℝ _ x y z)
#check (@neg_lt ℝ _ x y)

end my_abs
```

- Eliminación de la disyunción con rcases

```
-- ---------------------------------------------------------------------
-- Ejercicio. Sea x un número real. Demostrar que si
--    x ≠ 0
```

```
-- entonces
--    x < 0 ∨ x > 0
 -- ---------------------------------------------------------------------

import data.real.basic

example
  {x : ℝ}
  (h : x ≠ 0)
  : x < 0 ∨ x > 0 :=
begin
  rcases lt_trichotomy x 0 with xlt | xeq | xgt,
  { left,
    exact xlt },
  { contradiction },
  { right,
    exact xgt },
end

-- Prueba
-- ======

/-
x : ℝ,
h : x ≠ 0
⊢ x < 0 ∨ x > 0
  >> rcases lt_trichotomy x 0 with xlt | xeq | xgt,
| | xlt : x < 0
| | ⊢ x < 0 ∨ x > 0
| |   >> { left,
| | ⊢ x < 0
| |   >>  exact xlt },
| h : x ≠ 0,
| xeq : x = 0
| ⊢ x < 0 ∨ x > 0
  >> { contradiction },
h : x ≠ 0,
xgt : 0 < x
⊢ x < 0 ∨ x > 0
  >> { right,
⊢ x > 0
  >>  exact xgt },
no goals
-/
```

```
-- Comentarios:
-- 1. La táctica (rcases h with h1 | h2 | h3) si el objetivo es (P ∨ Q ∨ R)
--    crea tres casos añadiéndole al primero la hipótesis (h1 : P), al
--    segundo (h2 : Q) y al tercero (h3 : R).
```

- CS de divisibilidad del producto

```
-- ---------------------------------------------------------------------
-- Ejercicio. Demostrar que si m divide a n o a k, entonces divide a
-- m * k.
-- ---------------------------------------------------------------------

import tactic

variables {m n k : ℕ}

-- 1ª demostración
-- ===============

example
  (h : m ∣ n ∨ m ∣ k)
  : m ∣ n * k :=
begin
  rcases h with h1 | h2,
  { exact dvd_mul_of_dvd_left h1 k },
  { exact dvd_mul_of_dvd_right h2 n },
end

-- Prueba
-- ======

/-
m n k : ℕ,
h : m ∣ n ∨ m ∣ k
⊢ m ∣ n * k
  >> rcases h with h1 | h2,
| h1 : m ∣ n
| ⊢ m ∣ n * k
|   >> { exact dvd_mul_of_dvd_left h1 k },
h2 : m ∣ k
⊢ m ∣ n * k
  >> { exact dvd_mul_of_dvd_right h2 n },
no goals
-/
```

```
-- Comentario: Se han usado los lemas
-- + dvd_mul_of_dvd_left : m | n → ∀ (c : ℕ), m | n * c
-- + dvd_mul_of_dvd_right : m | n → ∀ (c : ℕ), m | c * n

-- Comprobación
#check (@dvd_mul_of_dvd_left ℕ _ m n)
#check (@dvd_mul_of_dvd_right ℕ _ m n)

-- 2ª demostración
-- ===============

example
  {m n k : ℕ}
  (h : m ∣ n ∨ m ∣ k)
  : m ∣ n * k :=
begin
  rcases h with ⟨a, rfl⟩ | ⟨b, rfl⟩,
  { rw [mul_assoc],
    apply dvd_mul_right },
  rw [mul_comm, mul_assoc],
  apply dvd_mul_right,
end

-- Prueba
-- ======

/-
m n k : ℕ,
h : m ∣ n ∨ m ∣ k
⊢ m ∣ n * k
  >> rcases h with ⟨a, rfl⟩ | ⟨b, rfl⟩,
| m k a : ℕ
| ⊢ m ∣ m * a * k
|   >> { rw [mul_assoc],
| ⊢ m ∣ m * (a * k)
|   >>   apply dvd_mul_right },
m n b : ℕ
⊢ m ∣ n * (m * b)
  >> rw [mul_comm, mul_assoc],
⊢ m ∣ m * (b * n)
  >> apply dvd_mul_right
no goals
-/
```

```
-- Comentario: Se han usado los siguientes lemas:
-- + mul_assoc m n k : m * n * k = m * (n * k)
-- + dvd_mul_right m n : m ∣ m * n
-- + mul_comm m n : m * n = n * m

-- Comprobación
#check (@mul_assoc ℕ _ m n k)
#check (@dvd_mul_right ℕ _ m n)
#check (@mul_comm ℕ _ m n)
```

- Desigualdad con rcases

```
-- ----------------------------------------------------------------------
-- Ejercicio. Demostrar que si
--    ∃ x y, z = x^2 + y^2 ∨ z = x^2 + y^2 + 1
-- entonces
--    z ≥ 0
-- ----------------------------------------------------------------------

import data.real.basic
import tactic

variables {z : ℝ}

example
  (h : ∃ x y, z = x^2 + y^2 ∨ z = x^2 + y^2 + 1)
  : z ≥ 0 :=
begin
  rcases h with ⟨a, b, h1 | h2⟩,
  { rw h1,
    apply add_nonneg,
    apply pow_two_nonneg,
    apply pow_two_nonneg },
  { rw h2,
    apply add_nonneg,
    apply add_nonneg,
    apply pow_two_nonneg,
    apply pow_two_nonneg,
    exact zero_le_one },
end

-- Prueba
-- ======
```

```
/-
z : ℝ,
h : ∃ (x y : ℝ), z = x ^ 2 + y ^ 2 ∨ z = x ^ 2 + y ^ 2 + 1
⊢ z ≥ 0
  >> rcases h with ⟨a, b, h1 | h2⟩,
| z a b : ℝ,
| h1 : z = a ^ 2 + b ^ 2
| ⊢ z ≥ 0
|   >> { rw h1,
| ⊢ a ^ 2 + b ^ 2 ≥ 0
|   >>   apply add_nonneg,
| | ⊢ 0 ≤ a ^ 2
|   >>   apply pow_two_nonneg,
| ⊢ 0 ≤ b ^ 2
|   >>   apply pow_two_nonneg },
h2 : z = a ^ 2 + b ^ 2 + 1
⊢ z ≥ 0
  >> { rw h2,
⊢ a ^ 2 + b ^ 2 + 1 ≥ 0
  >>   apply add_nonneg,
| ⊢ 0 ≤ a ^ 2 + b ^ 2
|   >>   apply add_nonneg,
| | ⊢ 0 ≤ a ^ 2
| |  >>   apply pow_two_nonneg,
| ⊢ 0 ≤ b ^ 2
|   >>   apply pow_two_nonneg,
⊢ 0 ≤ 1
  >>   exact zero_le_one },
no goals
-/

-- Comentarios:
-- 1. La táctica (rcases h with ⟨a, b, h1 | h2⟩) sobre el objetivo
--    (∃ x y : ℝ, P ∨ Q) crea dos casos. Al primero le añade las
--    hipótesis (a b : ℝ) y (k1 : P). Al segundo, (a b : ℝ) y (h2 : Q).
-- 2. Se han usado los siguientes lemas:
--    + add_nonneg : 0 ≤ x → 0 ≤ y → 0 ≤ x + y
--    + pow_two_nonneg x : 0 ≤ x ^ 2
--    + zero_le_one : 0 ≤ 1

-- Comprobación:
variables (x y : ℝ)
#check (@add_nonneg ℝ _ x y)
#check (@pow_two_nonneg ℝ _ x)
#check zero_le_one
```

- Igualdad de cuadrados

```
-- ----------------------------------------------------------------------
-- Ejercicio 1. Realizar las siguientes acciones:
-- 1. Importar la teoría de números reales.
-- 2. Declarar x e y como variables sobre los reales.
-- ----------------------------------------------------------------------

import data.real.basic   -- 1
variables (x y : ℝ)      -- 2

-- ----------------------------------------------------------------------
-- Ejercicio 2. Demostrar que si
--    x^2 = 1
-- entonces
--    x = 1 ∨ x = -1
-- ----------------------------------------------------------------------

example
  (h : x^2 = 1)
  : x = 1 ∨ x = -1 :=
begin
  have h1 : (x - 1) * (x + 1) = 0,
    calc (x - 1) * (x + 1) = x^2 - 1 : by ring
                       ... = 1 - 1   : by rw h
                       ... = 0       : by ring,
  have h2 : x - 1 = 0 ∨ x + 1 = 0,
    { apply eq_zero_or_eq_zero_of_mul_eq_zero h1 },
  cases h2,
  { left,
    exact sub_eq_zero.mp h2 },
  { right,
    exact eq_neg_of_add_eq_zero h2 },
end

-- Prueba
-- ======

/-
x : ℝ,
h : x ^ 2 = 1
⊢ x = 1 ∨ x = -1
  >> have h1 : (x - 1) * (x + 1) = 0,
  >>   calc (x - 1) * (x + 1) = x^2 - 1 : by ring
  >>                      ... = 1 - 1   : by rw h
```

```
  >>                        ... = 0       : by ring,
h1 : (x - 1) * (x + 1) = 0
⊢ x = 1 ∨ x = -1
  >> have h2 : x - 1 = 0 ∨ x + 1 = 0,
  >>   { apply eq_zero_or_eq_zero_of_mul_eq_zero h1 },
h2 : x - 1 = 0 ∨ x + 1 = 0
⊢ x = 1 ∨ x = -1
  >> cases h2,
| h2 : x - 1 = 0
| ⊢ x = 1 ∨ x = -1
|   >> { left,
| ⊢ x = 1
|   >>   exact sub_eq_zero.mp h2 },
h2 : x + 1 = 0
⊢ x = 1 ∨ x = -1
  >> { right,
⊢ x = -1
  >>   exact eq_neg_of_add_eq_zero h2 },
no goals
-/

-- Comentario: Se han usado los siguientes lemas
-- + eq_zero_or_eq_zero_of_mul_eq_zero : x * y = 0 → x = 0 ∨ y = 0
-- + sub_eq_zero : x - y = 0 ↔ x = y
-- + eq_neg_of_add_eq_zero : x + y = 0 → x = -y

-- Comprobación
#check (@eq_zero_or_eq_zero_of_mul_eq_zero ℝ _ _ _ x y)
#check (@sub_eq_zero ℝ _ x y)
#check (@eq_neg_of_add_eq_zero ℝ _ x y)

-- ---------------------------------------------------------------------
-- Ejercicio. Demostrar si
--    x^2 = y^2
-- entonces
--    x = y ∨ x = -y
-- ---------------------------------------------------------------------

example
  (h : x^2 = y^2)
  : x = y ∨ x = -y :=
begin
  have h1 : (x - y) * (x + y) = 0,
    calc (x - y) * (x + y) = x^2 - y^2 : by ring
                       ... = y^2 - y^2 : by rw h
```

```
                      ... = 0         : by ring,
  have h2 : x - y = 0 ∨ x + y = 0,
    { apply eq_zero_or_eq_zero_of_mul_eq_zero h1 },
  cases h2,
  { left,
    exact sub_eq_zero.mp h2 },
  { right,
    exact eq_neg_of_add_eq_zero h2 },
end

-- Comentario: Se han usado los siguientes lemas
-- + eq_zero_or_eq_zero_of_mul_eq_zero : x * y = 0 → x = 0 ∨ y = 0
-- + sub_eq_zero : x - y = 0 ↔ x = y
-- + eq_neg_of_add_eq_zero : x + y = 0 → x = -y
```

- Igualdad de cuadrados en dominios de integridad

```
-- ---------------------------------------------------------------------
-- Ejercicio. Importar las teorías:
-- + algebra.group_power de potencias en grupos
-- + tactic de tácticas
-- ---------------------------------------------------------------------

import algebra.group_power
import tactic

-- ---------------------------------------------------------------------
-- Ejercicio. Declara R como una variable sobre dominios de integridad.
-- ---------------------------------------------------------------------

variables {R : Type*} [integral_domain R]

-- ---------------------------------------------------------------------
-- Ejercicio. Declarar x e y como variables sobre R.
-- ---------------------------------------------------------------------

variables (x y : R)

-- ---------------------------------------------------------------------
-- Ejercicio. Demostrar si
--    x^2 = 1
-- entonces
--    x = 1 ∨ x = -1
-- ---------------------------------------------------------------------
```

```
example
  (h : x^2 = 1)
  : x = 1 ∨ x = -1 :=
begin
  have h1 : (x - 1) * (x + 1) = 0,
    calc (x - 1) * (x + 1) = x^2 - 1 : by ring
                       ... = 1 - 1   : by rw h
                       ... = 0       : by ring,
  have h2 : x - 1 = 0 ∨ x + 1 = 0,
    { apply eq_zero_or_eq_zero_of_mul_eq_zero h1 },
  cases h2,
  { left,
    exact sub_eq_zero.mp h2 },
  { right,
    exact eq_neg_of_add_eq_zero h2 },
end

-- ---------------------------------------------------------------------
-- Ejercicio. Demostrar si
--    x^2 = y^2
-- entonces
--    x = y ∨ x = -y
-- ---------------------------------------------------------------------

example
  (h : x^2 = y^2)
  : x = y ∨ x = -y :=
begin
  have h1 : (x - y) * (x + y) = 0,
    calc (x - y) * (x + y) = x^2 - y^2 : by ring
                       ... = y^2 - y^2 : by rw h
                       ... = 0         : by ring,
  have h2 : x - y = 0 ∨ x + y = 0,
    { apply eq_zero_or_eq_zero_of_mul_eq_zero h1 },
  cases h2,
  { left,
    exact sub_eq_zero.mp h2 },
  { right,
    exact eq_neg_of_add_eq_zero h2 },
end
```

- Eliminación de la doble negación (Tácticas çases emz "by_cases")

```
-- ---------------------------------------------------------------------
-- Ejercicio. Importar la librería de tácticas
-- ---------------------------------------------------------------------

import tactic

-- ---------------------------------------------------------------------
-- Ejercicio. Demostrar que
--    ¬ ¬ P → P
-- ---------------------------------------------------------------------

-- 1ª demostración
-- ===============

example (P : Prop) : ¬ ¬ P → P :=
begin
  intro h,
  cases classical.em P,
  { assumption },
  { contradiction },
end

-- Prueba
-- ======

/-
P : Prop
⊢ ¬¬P → P
  >> intro h,
h : ¬¬P
⊢ P
  >> cases classical.em P,
| h : ¬¬P
| ⊢ P
|   >> { assumption },
h_1 : ¬P
⊢ P
  >> { contradiction },
no goals
-/

-- 2ª demostración
-- ===============

open classical
```

```
example (P : Prop) : ¬ ¬ P → P :=
begin
  intro h,
  cases em P,
  { assumption },
  { contradiction },
end

-- 3ª demostración
-- ===============

open_locale classical

example (P : Prop) : ¬ ¬ P → P :=
begin
  intro h,
  by_cases h' : P,
  { assumption },
  { contradiction },
end

-- Prueba
-- ======

/-
P : Prop
⊢ ¬¬P → P
  >> intro h,
h : ¬¬P
⊢ P
  >> by_cases h' : P,
| 2 goals
| P : Prop,
| h : ¬¬P,
| h' : P
| ⊢ P
|   >> { assumption },
h' : ¬P
⊢ P
  >> { contradiction },
no goals
-/
```

- Implicación mediante disyunción y negación

```
-- ----------------------------------------------------------------------
-- Ejercicio. Demostrar que
--    (P → Q) ↔ ¬ P ∨ Q
-- ----------------------------------------------------------------------

import tactic

open_locale classical

-- 1ª demostración
-- ===============

example
  (P Q : Prop)
  : (P → Q) ↔ ¬ P ∨ Q :=
begin
  split,
  { intro h1,
    by_cases h2: P,
    { right,
      apply h1,
      exact h2 },
    { left,
      exact h2 }},
  { intros h3 h4,
    cases h3,
    { contradiction },
    { assumption }},
end

-- Prueba
-- ======

/-
P Q : Prop
⊢ P → Q ↔ ¬P ∨ Q
  >> split,
| 2 goals
| P Q : Prop
| ⊢ (P → Q) → ¬P ∨ Q
|   >> { intro h1,
| h1 : P → Q
| ⊢ ¬P ∨ Q
|   >>   by_cases h2: P,
| | 2 goals
```

```
| | P Q : Prop,
| | h1 : P → Q,
| | h2 : P
| | ⊢ ¬P ∨ Q
| |   >>   { right,
| | ⊢ Q
| |   >>     apply h1,
| | ⊢ P
| |   >>     exact h2 },
| P Q : Prop,
| h1 : P → Q,
| h2 : ¬P
| ⊢ ¬P ∨ Q
|   >>   { left,
| ⊢ ¬P
|   >>     exact h2 }},
P Q : Prop
⊢ ¬P ∨ Q → P → Q
  >> { intros h3 h4,
h3 : ¬P ∨ Q,
h4 : P
⊢ Q
  >>   cases h3,
| 2 goals
| case or.inl
| P Q : Prop,
| h4 : P,
| h3 : ¬P
| ⊢ Q
|   >>   { contradiction },
case or.inr
P Q : Prop,
h4 : P,
h3 : Q
⊢ Q
  >>   { assumption }},
no goals
-/

-- 2ª demostración
-- ===============

example
  (P Q : Prop)
  : (P → Q) ↔ ¬ P ∨ Q :=
```

```
imp_iff_not_or

-- 3ª demostración
-- ===============

example
  (P Q : Prop)
  : (P → Q) ↔ ¬ P ∨ Q :=
by tauto
```

3.6. Sucesiones y convergencia

- Definicion de convergencia

```
-- ---------------------------------------------------------------------
-- Ejercicio. Definir la función
--     converges_to (ℕ → ℝ) → ℝ → Prop
-- tal que (converges_to s a) afirma que a es el límite de s.
-- ---------------------------------------------------------------------

import data.real.basic

def converges_to (s : ℕ → ℝ) (a : ℝ) :=
∀ ε > 0, ∃ N, ∀ n ≥ N, abs (s n - a) < ε

#print converges_to

-- Comentario: Al colocar el cursor sobre print se obtiene
--    def converges_to : (ℕ → ℝ) → ℝ → Prop :=
--    λ (s : ℕ → ℝ) (a : ℝ),
--      ∀ (ε : ℝ), ε > 0 → (∃ (N : ℕ), ∀ (n : ℕ),
--        n ≥ N → abs (s n - a) < ε)
```

- Demostración por extensionalidad (La táctica ext)

```
-- ---------------------------------------------------------------------
-- Ejercicio. Demostrar que
--    (λ x y : ℝ, (x + y)^2) = (λ x y : ℝ, x^2 + 2*x*y + y^2)
-- ---------------------------------------------------------------------

import data.real.basic
```

```
-- 1ª demostración
-- ===============

example : (λ x y : ℝ, (x + y)^2) = (λ x y : ℝ, x^2 + 2*x*y + y^2) :=
begin
  ext,
  ring,
end

-- Prueba
-- ======

/-
⊢ (λ (x y : ℝ), (x + y) ^ 2) = λ (x y : ℝ), x ^ 2 + 2 * x * y + y ^ 2
  >> ext,
⊢ (x + x_1) ^ 2 = x ^ 2 + 2 * x * x_1 + x_1 ^ 2
  >> ring,
no goals
-/

-- Comentario: La táctica ext transforma las conclusiones de la forma
-- (λ x, f x) = (λ x, g x) en f x = g x.


-- 2ª demostración
-- ===============

example : (λ x y : ℝ, (x + y)^2) = (λ x y : ℝ, x^2 + 2*x*y + y^2) :=
by { ext, ring }

-- 3ª demostración
-- ===============

example : (λ x y : ℝ, (x + y)^2) = (λ x y : ℝ, x^2 + 2*x*y + y^2) :=
by ring
```

- Demostración por congruencia (La táctica congr)

```
-- ---------------------------------------------------------------------
-- Ejercicio. Demostrar que
--    abs a = abs (a - b + b)
-- ---------------------------------------------------------------------
```

```
import data.real.basic

-- 1ª demostración
-- ===============

example
  (a b : ℝ)
  : abs a = abs (a - b + b) :=
begin
  congr,
  ring,
end

-- Prueba
-- ======

/-
a b : ℝ
⊢ abs a = abs (a - b + b)
  >> congr,
⊢ a = a - b + b
  >> ring,
no goals
-/

-- Comentario: La táctica cong sustituye una conclusión de la forma
-- A = B por las igualdades de sus subtérminos que no no iguales por
-- definición. Por ejemplo, sustituye la conclusión (x * f y = g w * f z)
-- por las conclusiones (x = g w) y (y = z).

-- 2ª demostración
-- ===============

example
  (a b : ℝ)
  : abs a = abs (a - b + b) :=
by { congr, ring }

-- 3ª demostración
-- ===============

example
  (a b : ℝ)
  : abs a = abs (a - b + b) :=
by ring
```

- Demostración por conversión (La táctica convert)

```
-- ---------------------------------------------------------------------
-- Ejercicio. Demostrar, para todo a ∈ ℝ, si
--    1 < a
-- entonces
--    a < a * a
-- ---------------------------------------------------------------------

import data.real.basic

example
  {a : ℝ}
  (h : 1 < a)
  : a < a * a :=
begin
  convert (mul_lt_mul_right _).2 h,
  { rw [one_mul] },
  { exact lt_trans zero_lt_one h },
end

-- Prueba
-- ======

/-
a : ℝ,
h : 1 < a
⊢ a < a * a
  >> convert (mul_lt_mul_right _).2 h,
| 2 goals
| a : ℝ,
| h : 1 < a
| ⊢ a = 1 * a
|   >> { rw [one_mul] },
a : ℝ,
h : 1 < a
⊢ 0 < a
  >> { exact lt_trans zero_lt_one h },
no goals
-/

-- Comentarios:
-- 1. La táctica (convert e) genera nuevos subojetivos cuya conclusiones
--    son las diferencias entre el tipo de ge y la conclusión.
-- 2. Se han usado los siguientes lemas:
```

```
--    + mul_lt_mul_right : 0 < c → (a * c < b * c ↔ a < b)
--    + one_mul a : 1 * a = a
--    + lt_trans : a < b → b < c → a < c
--    + zero_lt_one : 0 < 1

-- Comprobación:
variables (a b c : ℝ)
#check @mul_lt_mul_right _ _ a b c
#check @one_mul _ _ a
#check @lt_trans _ _ a b c
#check @zero_lt_one _ _
```

- Convergencia de la función constante

```
-- ----------------------------------------------------------------------
-- Ejercicio. Demostrar que, para todo a ∈ ℝ, la función constante
--    s(x) = a
-- converge a a.
-- ----------------------------------------------------------------------

import .Definicion_de_convergencia

lemma converges_to_const
  (a : ℝ)
  : converges_to (λ x : ℕ, a) a :=
begin
  intros ε εpos,
  use 0,
  intros n nge,
  dsimp,
  rw sub_self,
  rw abs_zero,
  exact εpos,
end

-- Prueba
-- ======

/-
a : ℝ
⊢ converges_to (λ (x : ℕ), a) a
  >> intros ε εpos,
a ε : ℝ,
εpos : ε > 0
```

```
⊢ ∃ (N : ℕ), ∀ (n : ℕ), n ≥ N → abs ((λ (x : ℕ), a) n - a) < ε
  >> use 0,
⊢ ∀ (n : ℕ), n ≥ 0 → abs ((λ (x : ℕ), a) n - a) < ε
  >> intros n nge,
n : ℕ,
nge : n ≥ 0
⊢ abs ((λ (x : ℕ), a) n - a) < ε
  >> dsimp,
⊢ abs (a - a) < ε
  >> rw sub_self,
⊢ abs 0 < ε
  >> rw abs_zero,
⊢ abs 0 < ε
  >> exact εpos,
no goals
-/

-- Comentario: Se han usado los lemas
-- + sub_self a : a - a = 0
-- + abs_zero : abs 0 = 0

variables (a : ℝ)
#check @sub_self _ _ a
#check @abs_zero _ _
```

- Convergencia de la suma

```
-- ---------------------------------------------------------------------
-- Ejercicio. Demostrar el límite de la suma de dos sucesiones
-- convergentes es la suma de los límites.
-- ---------------------------------------------------------------------

import .Definicion_de_convergencia

variables {s t : ℕ → ℝ} {a b c : ℝ}

lemma converges_to_add
  (cs : converges_to s a)
  (ct : converges_to t b)
  : converges_to (λ n, s n + t n) (a + b) :=
begin
  intros ε εpos,
  dsimp,
  have ε2pos : 0 < ε / 2,
```

```
    { linarith },
  cases cs (ε / 2) ε2pos with Ns hs,
  cases ct (ε / 2) ε2pos with Nt ht,
  clear cs ct ε2pos εpos,
  use max Ns Nt,
  intros n hn,
  have nNs : n ≥ Ns,
    { exact le_of_max_le_left hn },
  specialize hs n nNs,
  have nNt : n ≥ Nt,
    { exact le_of_max_le_right hn },
  specialize ht n nNt,
  clear hn nNs nNt Ns Nt,
  calc abs (s n + t n - (a + b))
         = abs ((s n - a) + (t n -  b))   : by { congr, ring }
     ... ≤ abs (s n - a) + abs (t n -  b) : by apply abs_add
     ... < ε / 2 + ε / 2                  : by linarith [hs, ht]
     ... = ε                              : by apply add_halves,
end

-- Prueba
-- ======

/-
s t : ℕ → ℝ,
a b : ℝ,
cs : converges_to s a,
ct : converges_to t b
⊢ converges_to (λ (n : ℕ), s n + t n) (a + b)
  >> intros ε εpos,
ε : ℝ,
εpos : ε > 0
⊢ ∃ (N : ℕ), ∀ (n : ℕ), n ≥ N → abs ((λ (n : ℕ), s n + t n) n - (a + b)) < ε
  >> dsimp,
⊢ ∃ (N : ℕ), ∀ (n : ℕ), n ≥ N → abs (s n + t n - (a + b)) < ε
  >> have ε2pos : 0 < ε / 2,
| 2 goals
| s t : ℕ → ℝ,
| a b : ℝ,
| cs : converges_to s a,
| ct : converges_to t b,
| ε : ℝ,
| εpos : ε > 0
| ⊢ 0 < ε / 2
|   >>   { linarith },
```

```
s t : ℕ → ℝ,
a b : ℝ,
cs : converges_to s a,
ct : converges_to t b,
ε : ℝ,
εpos : ε > 0,
ε2pos : 0 < ε / 2
⊢ ∃ (N : ℕ), ∀ (n : ℕ), n ≥ N → abs (s n + t n - (a + b)) < ε
  >> cases cs (ε / 2) ε2pos with Ns hs,
ε2pos : 0 < ε / 2,
Ns : ℕ,
hs : ∀ (n : ℕ), n ≥ Ns → abs (s n - a) < ε / 2
⊢ ∃ (N : ℕ), ∀ (n : ℕ), n ≥ N → abs (s n + t n - (a + b)) < ε
  >> cases ct (ε / 2) ε2pos with Nt ht,
Nt : ℕ,
ht : ∀ (n : ℕ), n ≥ Nt → abs (t n - b) < ε / 2
⊢ ∃ (N : ℕ), ∀ (n : ℕ), n ≥ N → abs (s n + t n - (a + b)) < ε
  >> clear cs ct ε2pos εpos,
s t : ℕ → ℝ,
a b ε : ℝ,
Ns : ℕ,
hs : ∀ (n : ℕ), n ≥ Ns → abs (s n - a) < ε / 2,
Nt : ℕ,
ht : ∀ (n : ℕ), n ≥ Nt → abs (t n - b) < ε / 2
⊢ ∃ (N : ℕ), ∀ (n : ℕ), n ≥ N → abs (s n + t n - (a + b)) < ε
  >> use max Ns Nt,
⊢ ∀ (n : ℕ), n ≥ max Ns Nt → abs (s n + t n - (a + b)) < ε
  >> intros n hn,
n : ℕ,
hn : n ≥ max Ns Nt
⊢ abs (s n + t n - (a + b)) < ε
  >> have nNs : n ≥ Ns,
| 2 goals
| s t : ℕ → ℝ,
| a b ε : ℝ,
| Ns : ℕ,
| hs : ∀ (n : ℕ), n ≥ Ns → abs (s n - a) < ε / 2,
| Nt : ℕ,
| ht : ∀ (n : ℕ), n ≥ Nt → abs (t n - b) < ε / 2,
| n : ℕ,
| hn : n ≥ max Ns Nt
| ⊢ n ≥ Ns
|   >>   { exact le_of_max_le_left hn },
nNs : n ≥ Ns
⊢ abs (s n + t n - (a + b)) < ε
```

```
  >> specialize hs n nNs,
hs : abs (s n - a) < ε / 2
⊢ abs (s n + t n - (a + b)) < ε
  >> have nNt : n ≥ Nt,
| 2 goals
| s t : ℕ → ℝ,
| a b ε : ℝ,
| Ns Nt : ℕ,
| ht : ∀ (n : ℕ), n ≥ Nt → abs (t n - b) < ε / 2,
| n : ℕ,
| hn : n ≥ max Ns Nt,
| nNs : n ≥ Ns,
| hs : abs (s n - a) < ε / 2
| ⊢ n ≥ Nt
|   >>   { exact le_of_max_le_right hn },
s t : ℕ → ℝ,
a b ε : ℝ,
Ns Nt : ℕ,
ht : ∀ (n : ℕ), n ≥ Nt → abs (t n - b) < ε / 2,
n : ℕ,
hn : n ≥ max Ns Nt,
nNs : n ≥ Ns,
hs : abs (s n - a) < ε / 2,
nNt : n ≥ Nt
⊢ abs (s n + t n - (a + b)) < ε
  >> specialize ht n nNt,
ht : abs (t n - b) < ε / 2
⊢ abs (s n + t n - (a + b)) < ε
  >> clear hn nNs nNt Ns Nt,
s t : ℕ → ℝ,
a b ε : ℝ,
n : ℕ,
hs : abs (s n - a) < ε / 2,
ht : abs (t n - b) < ε / 2
⊢ abs (s n + t n - (a + b)) < ε
  >> calc abs (s n + t n - (a + b))
  >>          = abs ((s n - a) + (t n -  b))   : by { congr, ring }
  >>      ... ≤ abs (s n - a) + abs (t n -  b) : by apply abs_add
  >>      ... < ε / 2 + ε / 2                  : by linarith [hs, ht]
  >>      ... = ε                              : by apply add_halves,
no goals
-/

-- Comentario. Se han usado los lemas:
-- + le_of_max_le_left : max a b ≤ c → a ≤ c
```

```
-- + le_of_max_le_right : max a b ≤ c → b ≤ c
-- + abs_add a b : abs (a + b) ≤ abs a + abs b
-- + add_halves a : a / 2 + a / 2 = a

-- Comprobación
#check @le_of_max_le_left _ _ a b c
#check @le_of_max_le_right _ _ a b c
#check @abs_add _ _ a b
#check @add_halves _ _ a
```

- Convergencia del producto por una constante

```
-- ---------------------------------------------------------------------
-- Ejercicio. Demostrar si s es una sucesión que converge a a y c es un
-- número real, entonces c * s converge a c * a.
-- ---------------------------------------------------------------------

import .Convergencia_de_la_funcion_constante

variables {s : ℕ → ℝ} {a : ℝ}

lemma converges_to_mul_const_l1
  {c ε : ℝ}
  (h : 0 < c)
  : c * (ε / c) = ε :=
begin
  rw mul_comm,
  apply div_mul_cancel ε,
  exact ne_of_gt h,
end

-- Prueba
-- ======

/-
c ε : ℝ,
h : 0 < c
⊢ c * (ε / c) = ε
  >> rw mul_comm,
⊢ ε / c * c = ε
  >> apply div_mul_cancel ε,
⊢ c ≠ 0
  >> exact ne_of_gt h,
no goals
```

```
-/

theorem converges_to_mul_const
  {c : ℝ}
  (cs : converges_to s a)
  : converges_to (λ n, c * s n) (c * a) :=
begin
  by_cases h : c = 0,
  { convert converges_to_const 0,
    { ext,
      rw [h, zero_mul] },
    { rw [h, zero_mul] }},
  have acpos : 0 < abs c,
    from abs_pos_of_ne_zero h,
  intros ε εpos,
  dsimp,
  have εcpos : 0 < ε / abs c,
    by exact div_pos εpos acpos,
  cases cs (ε / abs c) εcpos with Ns hs,
  use Ns,
  intros n hn,
  specialize hs n hn,
  calc abs (c * s n - c * a)
         = abs (c * (s n - a))   : by { congr, ring }
     ... = abs c * abs (s n - a) : by apply abs_mul
     ... < abs c * (ε / abs c)   : by exact mul_lt_mul_of_pos_left hs acpos
     ... = ε                     : by apply converges_to_mul_const_l1 acpos
end

-- Prueba
-- ======

/-
s : ℕ → ℝ,
a c : ℝ,
cs : converges_to s a
⊢ converges_to (λ (n : ℕ), c * s n) (c * a)
  >> by_cases h : c = 0,
| h : c = 0
| ⊢ converges_to (λ (n : ℕ), c * s n) (c * a)
|   >> { convert converges_to_const 0,
| | ⊢ (λ (n : ℕ), c * s n) = λ (x : ℕ), 0
| |   >>   { ext,
| | x : ℕ
| | ⊢ c * s x = 0
```

```
| |   >>     rw [h, zero_mul] },
| s : ℕ → ℝ,
| a c : ℝ,
| cs : converges_to s a,
| h : c = 0
| ⊢ c * a = 0
|   >>   rw [h, zero_mul] },
s : ℕ → ℝ,
a c : ℝ,
cs : converges_to s a,
h : ¬c = 0
⊢ converges_to (λ (n : ℕ), c * s n) (c * a)
  >> have acpos : 0 < abs c,
| ⊢ 0 < abs c
|   >>   from abs_pos_of_ne_zero h,
acpos : 0 < abs c
⊢ converges_to (λ (n : ℕ), c * s n) (c * a)
  >> intros ε εpos,
ε : ℝ,
εpos : ε > 0
⊢ ∃ (N : ℕ), ∀ (n : ℕ), n ≥ N → abs ((λ (n : ℕ), c * s n) n - c * a) < ε
  >> dsimp,
⊢ ∃ (N : ℕ), ∀ (n : ℕ), n ≥ N → abs (c * s n - c * a) < ε
  >> have εcpos : 0 < ε / abs c,
  >>   by exact div_pos εpos acpos,
εcpos : 0 < ε / abs c
⊢ ∃ (N : ℕ), ∀ (n : ℕ), n ≥ N → abs (c * s n - c * a) < ε
  >> cases cs (ε / abs c) εcpos with Ns hs,
Ns : ℕ,
hs : ∀ (n : ℕ), n ≥ Ns → abs (s n - a) < ε / abs c
⊢ ∃ (N : ℕ), ∀ (n : ℕ), n ≥ N → abs (c * s n - c * a) < ε
  >> use Ns,
⊢ ∀ (n : ℕ), n ≥ Ns → abs (c * s n - c * a) < ε
  >> intros n hn,
n : ℕ,
hn : n ≥ Ns
⊢ abs (c * s n - c * a) < ε
  >> specialize hs n hn,
hs : abs (s n - a) < ε / abs c
⊢ abs (c * s n - c * a) < ε
  >> calc abs (c * s n - c * a)
  >>     = abs (c * (s n - a))   : by { congr, ring }
  >> ... = abs c * abs (s n - a) : by apply abs_mul
  >> ... < abs c * (ε / abs c)   : by exact mul_lt_mul_of_pos_left hs acpos
  >> ... = ε                     : by apply converges_to_mul_const_l1-/
```

```
/-acpos,
no goals
-/

-- Comentario: Se han usado los lemas
-- + mul_comm a b : a * b = b * a
-- + ne_of_gt : a > b → a ≠ b
-- + converges_to_const a : converges_to (λ (x : ℕ), a) a
-- + zero_mul a : 0 * a = 0
-- + abs_pos_of_ne_zero : a ≠ 0 → 0 < abs a
-- + div_pos : 0 < a → 0 < b → 0 < a / b
-- + abs_mul a b : abs (a * b) = abs a * abs b
-- + mul_lt_mul_of_pos_left : a < b → 0 < d → d * a < d * b

-- Comprobación:
variables (b d : ℝ)
#check @mul_comm _ _ a b
#check @ne_of_gt _ _ a b
#check converges_to_const a
#check @zero_mul _ _ a
#check @abs_pos_of_ne_zero _ _ a
#check @div_pos _ _ a b
#check @abs_mul _ _ a b
#check @mul_lt_mul_of_pos_left _ _ a b d
```

- Acotación de convergentes

```
-- ---------------------------------------------------------------------
-- Ejercicio. Demostrar si a es el límite de s, entonces
--     ∃ N b, ∀ n, N ≤ n → abs (s n) < b :=
-- ---------------------------------------------------------------------

import .Definicion_de_convergencia

variables {s : ℕ → ℝ} {a : ℝ}

theorem exists_abs_le_of_converges_to
  (cs : converges_to s a)
  : ∃ N b, ∀ n, N ≤ n → abs (s n) < b :=
begin
  cases cs 1 zero_lt_one with N h,
  use [N, abs a + 1],
  intros n hn,
  specialize h n hn,
```

```
  calc abs (s n)
          = abs (s n - a + a)     : by ring
      ... ≤ abs (s n - a) + abs a : by apply abs_add_le_abs_add_abs
      ... < 1 + abs a             : by exact add_lt_add_right h (abs a)
      ... = abs a + 1             : by exact add_comm 1 (abs a),
end

-- Prueba
-- ======

/-
s : ℕ → ℝ,
a : ℝ,
cs : converges_to s a
⊢ ∃ (N : ℕ) (b : ℝ), ∀ (n : ℕ), N ≤ n → abs (s n) < b
  >> cases cs 1 zero_lt_one with N h,
N : ℕ,
h : ∀ (n : ℕ), n ≥ N → abs (s n - a) < 1
⊢ ∃ (N : ℕ) (b : ℝ), ∀ (n : ℕ), N ≤ n → abs (s n) < b
  >> use [N, abs a + 1],
⊢ ∀ (n : ℕ), N ≤ n → abs (s n) < abs a + 1
  >> intros n hn,
n : ℕ,
hn : N ≤ n
⊢ abs (s n) < abs a + 1
  >> specialize h n hn,
h : abs (s n - a) < 1
⊢ abs (s n) < abs a + 1
  >> calc abs (s n)
  >>          = abs (s n - a + a)     : by ring
  >>      ... ≤ abs (s n - a) + abs a : by apply abs_add_le_abs_add_abs
  >>      ... < 1 + abs a             : by exact add_lt_add_right h (abs a)
  >>      ... = abs a + 1             : by exact add_comm 1 (abs a),
no goals
-/

-- Comentario: Se usan los lemas
-- + zero_lt_one : 0 < 1
-- + abs_add_le_abs_add_abs a b : abs (a + b) ≤ abs a + abs b
-- + add_lt_add_right : a < b → ∀ (c : ℝ), a + c < b + c
-- + add_comm a b : a + b = b + a

-- Comprobación:
variable (b : ℝ)
#check @zero_lt_one _ _
```

```
#check @abs_add_le_abs_add_abs _ _ a b
#check @add_lt_add_right _ _ a b
#check @add_comm _ _ a b
```

- Producto por sucesión convergente a cero

```
-- ---------------------------------------------------------------------
-- Ejercicio. Demostrar si s es una sucesión convergente y el límite de
-- t es 0, entonces el límite de s * t es 0.
-- ---------------------------------------------------------------------

import .Acotacion_de_convergentes

variables {s t : ℕ → ℝ} {a : ℝ}

lemma aux_l1
  (B ε : ℝ)
  (εpos : ε > 0)
  (Bpos : 0 < B)
  (pos0 : ε / B > 0)
  (n : ℕ)
  (h0 : abs (s n) < B)
  (h1 : abs (t n - 0) < ε / B)
  : abs (s n) * abs (t n - 0) < ε :=
begin
  by_cases h3 : s n = 0,
  { calc abs (s n) * abs (t n - 0)
              = abs 0 * abs (t n - 0) : by rw h3
          ... = 0 * abs (t n - 0)     : by rw abs_zero
          ... = 0                     : by exact zero_mul (abs (t n - 0))
          ... < ε                     : by exact εpos },
  { have h4 : abs (s n) > 0,
      by exact abs_pos_iff.mpr h3,
    clear h3,
    have h5 : abs (s n) * abs (t n - 0) < abs (s n) * (ε / B),
      by exact mul_lt_mul_of_pos_left h1 h4,
    have h6 : abs (s n) * (ε / B) < B * (ε / B),
      by exact mul_lt_mul_of_pos_right h0 pos0,
    have h7 : B ≠ 0,
      by exact ne_of_gt Bpos,
    have h8 : B * (ε / B) = ε,
      calc B * (ε / B) = (B * B⁻¹) * ε : by ring
                   ... = 1 * ε         : by rw mul_inv_cancel h7
                   ... = ε             : by exact one_mul ε,
```

```
    have h9 : abs (s n) * abs (t n - 0) < B * (ε / B),
      by exact gt.trans h6 h5,
    rw h8 at h9,
    assumption },
end

-- Prueba
-- ======

/-
s t : ℕ → ℝ,
B ε : ℝ,
εpos : ε > 0,
Bpos : 0 < B,
pos0 : ε / B > 0,
n : ℕ,
h0 : abs (s n) < B,
h1 : abs (t n - 0) < ε / B
⊢ abs (s n) * abs (t n - 0) < ε
  >> by_cases h3 : s n = 0,
| h3 : s n = 0
| ⊢ abs (s n) * abs (t n - 0) < ε
|   >> { calc abs (s n) * abs (t n - 0)
|   >>            = abs 0 * abs (t n - 0) : by rw h3
|   >>        ... = 0 * abs (t n - 0)     : by rw abs_zero
|   >>        ... = 0                     : by exact zero_mul (abs (t n - 0))
|   >>        ... < ε                     : by exact εpos },
  >> { have h4 : abs (s n) > 0,
  >>     by exact abs_pos_iff.mpr h3,
h4 : abs (s n) > 0
⊢ abs (s n) * abs (t n - 0) < ε
  >>   clear h3,
  >>   have h5 : abs (s n) * abs (t n - 0) < abs (s n) * (ε / B),
  >>     by exact mul_lt_mul_of_pos_left h1 h4,
h5 : abs (s n) * abs (t n - 0) < abs (s n) * (ε / B)
⊢ abs (s n) * abs (t n - 0) < ε
  >>   have h6 : abs (s n) * (ε / B) < B * (ε / B),
  >>     by exact mul_lt_mul_of_pos_right h0 pos0,
h6 : abs (s n) * (ε / B) < B * (ε / B)
⊢ abs (s n) * abs (t n - 0) < ε
  >>   have h7 : B ≠ 0,
  >>     by exact ne_of_gt Bpos,
h7 : B ≠ 0
⊢ abs (s n) * abs (t n - 0) < ε
  >>   have h8 : B * (ε / B) = ε,
```

```
  >>     calc B * (ε / B) = (B * B⁻¹) * ε : by ring
  >>                  ... = 1 * ε         : by rw mul_inv_cancel h7
  >>                  ... = ε             : by exact one_mul ε,
h8 : B * (ε / B) = ε
⊢ abs (s n) * abs (t n - 0) < ε
  >>   have h9 : abs (s n) * abs (t n - 0) < B * (ε / B),
  >>     by exact gt.trans h6 h5,
h9 : abs (s n) * abs (t n - 0) < B * (ε / B)
⊢ abs (s n) * abs (t n - 0) < ε
  >>   rw h8 at h9,
h9 : abs (s n) * abs (t n - 0) < ε
⊢ abs (s n) * abs (t n - 0) < ε
  >>   assumption },
no goals
-/

-- Comentarios: Se han usado los lemas
-- + abs_zero : abs 0 = 0
-- + zero_mul a : 0 * a = 0
-- + abs_pos_iff : 0 < abs a ↔ a ≠ 0
-- + mul_lt_mul_of_pos_left : a < b → 0 < c → c * a < c * b
-- + mul_lt_mul_of_pos_right : a < b → 0 < c → a * c < b * c
-- + ne_of_gt : a > b → a ≠ b
-- + mul_inv_cancel : a ≠ 0 → a * a⁻¹ = 1
-- + one_mul a : 1 * a = a
-- + gt.trans : a > b → b > c → a > c

variables (b c : ℝ)
#check @abs_pos_iff _ _ a
#check abs_zero
#check @gt.trans _ _ a b c
#check @mul_inv_cancel _ _ a
#check @mul_lt_mul_of_pos_left _ _ a b c
#check @mul_lt_mul_of_pos_right _ _ a b c
#check @ne_of_gt _ _ a b
#check @one_mul _ _ a
#check @zero_mul _ _ a

lemma aux
  (cs : converges_to s a)
  (ct : converges_to t 0)
  : converges_to (λ n, s n * t n) 0 :=
begin
  intros ε εpos,
  dsimp,
```

```
  rcases exists_abs_le_of_converges_to cs with ⟨N₀, B, h₀⟩,
  have Bpos : 0 < B,
    from lt_of_le_of_lt (abs_nonneg _) (h₀ N₀ (le_refl _)),
  have pos₀ : ε / B > 0,
    from div_pos εpos Bpos,
  cases ct _ pos₀ with N₁ h₁,
  use [max N₀ N₁],
  intros n hn,
  have hn0 : n ≥ N₀,
    { exact le_of_max_le_left hn },
  specialize h₀ n hn0,
  have hn1 : n ≥ N₁,
    { exact le_of_max_le_right hn },
  specialize h₁ n hn1,
  clear cs ct hn hn0 hn1 a N₀ N₁,
  calc
    abs (s n * t n - 0)
        = abs (s n * (t n - 0))
              : by { congr, ring }
    ... = abs (s n) * abs (t n - 0)
              : by exact abs_mul (s n) (t n - 0)
    ... < ε
              : by exact aux_l1 B ε εpos Bpos pos₀ n h₀ h₁,
end

-- Prueba
-- ======

/-
s t : ℕ → ℝ,
a : ℝ,
cs : converges_to s a,
ct : converges_to t 0
⊢ converges_to (λ (n : ℕ), s n * t n) 0
  >> intros ε εpos,
ε : ℝ,
εpos : ε > 0
⊢ ∃ (N : ℕ), ∀ (n : ℕ), n ≥ N → abs ((λ (n : ℕ), s n * t n) n - 0) < ε
  >> dsimp,
⊢ ∃ (N : ℕ), ∀ (n : ℕ), n ≥ N → abs (s n * t n - 0) < ε
  >> rcases exists_abs_le_of_converges_to cs with ⟨N₀, B, h₀⟩,
N₀ : ℕ,
B : ℝ,
h₀ : ∀ (n : ℕ), N₀ ≤ n → abs (s n) < B
⊢ ∃ (N : ℕ), ∀ (n : ℕ), n ≥ N → abs (s n * t n - 0) < ε
```

```
  >> have Bpos : 0 < B,
  >>   from lt_of_le_of_lt (abs_nonneg _) (h₀ N₀ (le_refl _)),
Bpos : 0 < B
⊢ ∃ (N : ℕ), ∀ (n : ℕ), n ≥ N → abs (s n * t n - 0) < ε
  >> have pos₀ : ε / B > 0,
  >>   from div_pos εpos Bpos,
pos₀ : ε / B > 0
⊢ ∃ (N : ℕ), ∀ (n : ℕ), n ≥ N → abs (s n * t n - 0) < ε
  >> cases ct _ pos₀ with N₁ h₁,
N₁ : ℕ,
h₁ : ∀ (n : ℕ), n ≥ N₁ → abs (t n - 0) < ε / B
⊢ ∃ (N : ℕ), ∀ (n : ℕ), n ≥ N → abs (s n * t n - 0) < ε
  >> use [max N₀ N₁],
⊢ ∀ (n : ℕ), n ≥ max N₀ N₁ → abs (s n * t n - 0) < ε
  >> intros n hn,
n : ℕ,
hn : n ≥ max N₀ N₁
⊢ abs (s n * t n - 0) < ε
  >> have hn0 : n ≥ N₀,
  >>   { exact le_of_max_le_left hn },
hn0 : n ≥ N₀
⊢ abs (s n * t n - 0) < ε
  >> specialize h₀ n hn0,
h₀ : abs (s n) < B
⊢ abs (s n * t n - 0) < ε
  >> have hn1 : n ≥ N₁,
  >>   { exact le_of_max_le_right hn },
hn1 : n ≥ N₁
⊢ abs (s n * t n - 0) < ε
  >> specialize h₁ n hn1,
h₁ : abs (t n - 0) < ε / B
⊢ abs (s n * t n - 0) < ε
  >> clear cs ct hn hn0 hn1 a N₀ N₁,
  >> calc
  >>   abs (s n * t n - 0)
  >>       = abs (s n * (t n - 0))
  >>             : by { congr, ring }
  >>   ... = abs (s n) * abs (t n - 0)
  >>             : by exact abs_mul (s n) (t n - 0)
  >>   ... < ε
  >>             : by exact aux_l1 B ε εpos Bpos pos₀ n h₀ h₁,
no goals
-/

-- Comentarios: Se han usado los lemas
```

```
-- + abs_mul : abs (a * b) = abs a * abs b
-- + abs_nonneg a : 0 ≤ abs a
-- + div_pos : 0 < a → 0 < b → 0 < a / b
-- + le_of_max_le_left : max a b ≤ c → a ≤ c
-- + le_of_max_le_right : max a b ≤ c → b ≤ c
-- + lt_of_le_of_lt : a ≤ b → b < c → a < c

-- Comprobación:
variables (b c : ℝ)
#check @lt_of_le_of_lt _ _ a b c
#check @abs_nonneg _ _ a
#check @div_pos _ _ a b
#check @le_of_max_le_left _ _ a b c
#check @le_of_max_le_right _ _ a b c
#check abs_mul
```

- Convergencia del producto

```
-- ---------------------------------------------------------------------
-- Ejercicio. Demostrar el límite del producto de dos sucesiones
-- convergentes es el producto de sus límites.
-- ---------------------------------------------------------------------

import .Definicion_de_convergencia
import .Convergencia_de_la_funcion_constante
import .Convergencia_de_la_suma
import .Convergencia_del_producto_por_una_constante
import .Acotacion_de_convergentes
import .Producto_por_sucesion_convergente_a_cero
import tactic

variables {s t : ℕ → ℝ} {a b : ℝ}

theorem converges_to_mul
  (cs : converges_to s a)
  (ct : converges_to t b)
  : converges_to (λ n, s n * t n) (a * b) :=
begin
  have h₁ : converges_to (λ n, s n * (t n - b)) 0,
  { apply aux cs,
    convert converges_to_add ct (converges_to_const (-b)),
    ring },
  convert (converges_to_add h₁ (@converges_to_mul_const s a b cs)),
  { ext,
```

```
    ring },
  { ring },
end

-- Prueba
-- ======

/-
s t : ℕ → ℝ,
a b : ℝ,
cs : converges_to s a,
ct : converges_to t b
⊢ converges_to (λ (n : ℕ), s n * t n) (a * b)
  >> have h₁ : converges_to (λ n, s n * (t n - b)) 0,
| ⊢ converges_to (λ (n : ℕ), s n * (t n - b)) 0
|   >> { apply aux cs,
| ⊢ converges_to (λ (n : ℕ), t n - b) 0
|   >>   convert converges_to_add ct (converges_to_const (-b)),
| ⊢ 0 = b + -b
|   >>   ring },
h₁ : converges_to (λ (n : ℕ), s n * (t n - b)) 0
⊢ converges_to (λ (n : ℕ), s n * t n) (a * b)
  >> convert (converges_to_add h₁ (@converges_to_mul_const s a b cs)),
| ⊢ (λ (n : ℕ), s n * t n) = λ (n : ℕ), s n * (t n - b) + b * s n
|   >> { ext,
| x : ℕ
| ⊢ s x * t x = s x * (t x - b) + b * s x
|   >>   ring },
⊢ a * b = 0 + b * a
  >> { ring },
no goals
-/
```

- Unicidad del límite

```
-- ---------------------------------------------------------------------
-- Ejercicio. Demostrar la unicidad de los límites de las sucesiones
-- convergentes.
-- ---------------------------------------------------------------------

import .Definicion_de_convergencia

open_locale classical
```

```
theorem converges_to_unique
  {s : ℕ → ℝ}
  {a b : ℝ}
  (sa : converges_to s a)
  (sb : converges_to s b)
  : a = b :=
begin
  by_contradiction abne,
  have : abs (a - b) > 0,
  { apply abs_pos_of_ne_zero,
    exact sub_ne_zero_of_ne abne },
  let ε := abs (a - b) / 2,
  have εpos : ε > 0,
  { change abs (a - b) / 2 > 0,
    linarith },
  cases sa ε εpos with Na hNa,
  cases sb ε εpos with Nb hNb,
  let N := max Na Nb,
  have absa : abs (s N - a) < ε,
  { specialize hNa N,
    apply hNa,
    exact le_max_left Na Nb },
  have absb : abs (s N - b) < ε,
  { specialize hNb N,
    apply hNb,
    exact le_max_right Na Nb },
  have : abs (a - b) < abs (a - b),
    calc abs (a - b)
         = abs ((a - s N) + (s N - b))      : by {congr, ring}
     ... ≤ abs (a - s N) + abs (s N - b)    : by apply abs_add_le_abs_add_abs
     ... = abs (s N - a) + abs (s N - b)    : by rw abs_sub
     ... < ε + ε                            : by exact add_lt_add absa absb
     ... = abs (a - b)                      : by exact add_halves (abs (a - b)),
  exact lt_irrefl _ this,
end

-- Prueba
-- ======

/-
s : ℕ → ℝ,
a b : ℝ,
sa : converges_to s a,
sb : converges_to s b
⊢ a = b
```

```
  >> by_contradiction abne,
abne : ¬a = b
⊢ false
  >> have : abs (a - b) > 0,
| ⊢ abs (a - b) > 0
|   >> { apply abs_pos_of_ne_zero,
| ⊢ a - b ≠ 0
|   >>   exact sub_ne_zero_of_ne abne },
this : abs (a - b) > 0
⊢ false
  >> let ε := abs (a - b) / 2,
ε : ℝ := abs (a - b) / 2
⊢ false
  >> have εpos : ε > 0,
| ⊢ ε > 0
|   >> { change abs (a - b) / 2 > 0,
| ⊢ abs (a - b) / 2 > 0
|   >>   linarith },
εpos : ε > 0
⊢ false
  >> cases sa ε εpos with Na hNa,
Na : ℕ,
hNa : ∀ (n : ℕ), n ≥ Na → abs (s n - a) < ε
⊢ false
  >> cases sb ε εpos with Nb hNb,
Nb : ℕ,
hNb : ∀ (n : ℕ), n ≥ Nb → abs (s n - b) < ε
⊢ false
  >> let N := max Na Nb,
N : ℕ := max Na Nb
⊢ false
  >> have absa : abs (s N - a) < ε,
| ⊢ abs (s N - a) < ε
|   >> { specialize hNa N,
| hNa : N ≥ Na → abs (s N - a) < ε
| ⊢ abs (s N - a) < ε
|   >>   apply hNa,
| ⊢ N ≥ Na
|   >>   exact le_max_left Na Nb },
absa : abs (s N - a) < ε
⊢ false
  >> have absb : abs (s N - b) < ε,
| ⊢ abs (s N - b) < ε
|   >> { specialize hNb N,
| hNb : N ≥ Nb → abs (s N - b) < ε
```

```
| ⊢ abs (s N - b) < ε
|   >>   apply hNb,
| hNb : N ≥ Nb → abs (s N - b) < ε
|   >>   exact le_max_right Na Nb },
absb : abs (s N - b) < ε
⊢ false
  >> have : abs (a - b) < abs (a - b),
  >>   calc abs (a - b)
  >>        = abs ((a - s N) + (s N - b))   : by {congr, ring}
  >>    ... ≤ abs (a - s N) + abs (s N - b) : by apply abs_add_le_abs_add_abs
  >>    ... = abs (s N - a) + abs (s N - b) : by rw abs_sub
  >>    ... < ε + ε                         : by exact add_lt_add absa absb
  >>    ... = abs (a - b)                   : by exact add_halves (abs (a - b)),
this : abs (a - b) < abs (a - b)
⊢ false
  >> exact lt_irrefl _ this,
no goals
-/

-- Comentario: Se han usado los lemas
-- + abs_add_le_abs_add_abs a b : abs (a + b) ≤ abs a + abs b
-- + abs_pos_of_ne_zero : a ≠ 0 → 0 < abs a
-- + abs_sub a b : abs (a - b) = abs (b - a)
-- + add_halves a : a / 2 + a / 2 = a
-- + add_lt_add : a < b → c < d → a + c < b + d
-- + le_max_left a b : a ≤ max a b
-- + le_max_right a b : b ≤ max a b
-- + sub_ne_zero_of_ne : a ≠ b → a - b ≠ 0

-- Comprobación:
variables (a b c d : ℝ)
#check @abs_pos_of_ne_zero _ _ a
#check @sub_ne_zero_of_ne _ _ a b
#check @le_max_left _ _ a b
#check @le_max_right _ _ a b
#check @abs_add_le_abs_add_abs _ _ a b
#check @abs_sub _ _ a b
#check @add_lt_add _ _ a b c d
#check @add_halves _ _ a
```

Capítulo 4

Conjuntos y funciones

En este capítulo se muestra el razonamiento con Lean sobre las operaciones conjuntistas y sobre las funciones.

4.1. Conjuntos

- Monotonía de la intersección

```
-- ---------------------------------------------------------------------
-- Ejercicio. Demostrar si
--    s ⊆ t
-- entonces
--    s ∩ u ⊆ t ∩ u
-- ---------------------------------------------------------------------

import tactic

variable {α : Type*}
variables (s t u : set α)

open set

-- 1ª demostración
-- ===============

example
  (h : s ⊆ t)
  : s ∩ u ⊆ t ∩ u :=
begin
  rw subset_def,
```

```
  rw inter_def,
  rw inter_def,
  dsimp,
  rw subset_def at h,
  rintros x ⟨xs, xu⟩,
  split,
  { exact h x xs },
  { exact xu },
end

-- Prueba
-- ======

/-
α : Type u_1,
s t u : set α,
h : s ⊆ t
⊢ s ∩ u ⊆ t ∩ u
  >> rw subset_def,
⊢ ∀ (x : α), x ∈ s ∩ u → x ∈ t ∩ u
  >> rw inter_def,
⊢ ∀ (x : α), x ∈ {a : α | a ∈ s ∧ a ∈ u} → x ∈ t ∩ u
  >> rw inter_def,
⊢ ∀ (x : α), x ∈ {a : α | a ∈ s ∧ a ∈ u} → x ∈ {a : α | a ∈ t ∧ a ∈ u}
  >> dsimp,
⊢ ∀ (x : α), x ∈ s ∧ x ∈ u → x ∈ t ∧ x ∈ u
  >> rw subset_def at h,
h : ∀ (x : α), x ∈ s → x ∈ t
⊢ ∀ (x : α), x ∈ s ∧ x ∈ u → x ∈ t ∧ x ∈ u
  >> rintros x ⟨xs, xu⟩,
x : α,
xs : x ∈ s,
xu : x ∈ u
⊢ x ∈ t ∧ x ∈ u
  >> split,
| ⊢ x ∈ t
|   >> { exact h x xs },
⊢ x ∈ u
  >> { exact xu },
no goals
-/

-- 2ª demostración
-- ===============
```

```
example
  (h : s ⊆ t)
  : s ∩ u ⊆ t ∩ u :=
begin
  rw [subset_def, inter_def, inter_def],
  dsimp,
  rw subset_def at h,
  rintros x ⟨xs, xu⟩,
  exact ⟨h _ xs, xu⟩,
end

-- Prueba
-- ======

/-
α : Type u_1,
s t u : set α,
h : s ⊆ t
⊢ s ∩ u ⊆ t ∩ u  >> rw [subset_def, inter_def, inter_def],
  >> dsimp,
⊢ ∀ (x : α), x ∈ s ∧ x ∈ u → x ∈ t ∧ x ∈ u
  >> rw subset_def at h,
h : ∀ (x : α), x ∈ s → x ∈ t
⊢ ∀ (x : α), x ∈ s ∧ x ∈ u → x ∈ t ∧ x ∈ u
  >> rintros x ⟨xs, xu⟩,
x : α,
xs : x ∈ s,
xu : x ∈ u
⊢ x ∈ t ∧ x ∈ u
  >> exact ⟨h _ xs, xu⟩,
no goals
-/

-- 3ª demostración
-- ===============

example
  (h : s ⊆ t)
  : s ∩ u ⊆ t ∩ u :=
begin
  simp only [subset_def, mem_inter_eq] at *,
  rintros x ⟨xs, xu⟩,
  exact ⟨h _ xs, xu⟩,
end
```

```
-- Prueba
-- ======

/-
α : Type u_1,
s t u : set α,
h : s ⊆ t
⊢ s ∩ u ⊆ t ∩ u
  >> simp only [subset_def, mem_inter_eq] at *,
h : ∀ (x : α), x ∈ s → x ∈ t
⊢ ∀ (x : α), x ∈ s ∧ x ∈ u → x ∈ t ∧ x ∈ u
  >> rintros x ⟨xs, xu⟩,
x : α,
xs : x ∈ s,
xu : x ∈ u
⊢ x ∈ t ∧ x ∈ u
  >> exact ⟨h _ xs, xu⟩,
no goals
-/

-- 4ª demostración
-- ===============

example
  (h : s ⊆ t)
  : s ∩ u ⊆ t ∩ u :=
by finish [subset_def, mem_inter_eq]

-- 5ª demostración
-- ===============

example
  (h : s ⊆ t)
  : s ∩ u ⊆ t ∩ u :=
begin
  intros x xsu,
  exact ⟨h xsu.1, xsu.2⟩,
end

-- Prueba
-- ======

/-
α : Type u_1,
```

```
s t u : set α,
h : s ⊆ t
⊢ s ∩ u ⊆ t ∩ u
  >> intros x xsu,
x : α,
xsu : x ∈ s ∩ u
⊢ x ∈ t ∩ u
  >> exact ⟨h xsu.1, xsu.2⟩,
no goals
-/

-- Comentario: La táctica *intro* aplica una *reducción definicional*
-- expandiendo las definiciones.

-- 6ª demostración
-- ===============

example (h : s ⊆ t) : s ∩ u ⊆ t ∩ u :=
by exact λ x ⟨xs, xu⟩, ⟨h xs, xu⟩

-- 7ª demostración
-- ===============

lemma monotonia
  (h : s ⊆ t)
  : s ∩ u ⊆ t ∩ u :=
λ x ⟨xs, xu⟩, ⟨h xs, xu⟩

-- 8ª demostración
-- ===============

example
  (h : s ⊆ t)
  : s ∩ u ⊆ t ∩ u :=
inter_subset_inter_left u h

-- Comentario: Se han usado los lemas
-- + inter_def : s ∩ t = {a : α | a ∈ s ∧ a ∈ t}
-- + inter_subset_inter_left : s ⊆ t → s ∩ u ⊆ t ∩ u
-- + mem_inter_eq x s t : x ∈ s ∩ t = (x ∈ s ∧ x ∈ t)
-- + subset_def : s ⊆ t = ∀ (x : α), x ∈ s → x ∈ t

-- Comprobación:
variable (x : α)
#check @subset_def _ s t
```

```
#check @inter_def _ s t
#check @mem_inter_eq _ x s t
#check @inter_subset_inter_left _ s t u
```

- Distributiva de la intersección

```
import tactic

variable {α : Type*}
variables (s t u : set α)

-- ---------------------------------------------------------------------
-- Ejercicio. Demostrar que
--    s ∩ (t ∪ u) ⊆ (s ∩ t) ∪ (s ∩ u)
-- ---------------------------------------------------------------------

-- 1ª demostración
-- ===============

example : s ∩ (t ∪ u) ⊆ (s ∩ t) ∪ (s ∩ u) :=
begin
  intros x hx,
  have xs : x ∈ s := hx.1,
  have xtu : x ∈ t ∪ u := hx.2,
  cases xtu with xt xu,
  { left,
    show x ∈ s ∩ t,
    exact ⟨xs, xt⟩ },
  { right,
    show x ∈ s ∩ u,
    exact ⟨xs, xu⟩ },
end

-- Prueba
-- ======

/-
α : Type u_1,
s t u : set α
⊢ s ∩ (t ∪ u) ⊆ s ∩ t ∪ s ∩ u
  >> intros x hx,
x : α,
hx : x ∈ s ∩ (t ∪ u)
```

```
⊢ x ∈ s ∩ t ∪ s ∩ u
  >> have xs : x ∈ s := hx.1,
xs : x ∈ s
⊢ x ∈ s ∩ t ∪ s ∩ u
  >> have xtu : x ∈ t ∪ u := hx.2,
⊢ xtu : x ∈ t ∪ u
  >> cases xtu with xt xu,
| xt : x ∈ t
| ⊢ x ∈ s ∩ t ∪ s ∩ u
|   >> { left,
| ⊢ x ∈ s ∩ t
|   >>   show x ∈ s ∩ t,
| ⊢ x ∈ s ∩ t
|   >>   exact ⟨xs, xt⟩ },
xu : x ∈ u
⊢ x ∈ s ∩ t ∪ s ∩ u
  >> { right,
⊢ x ∈ s ∩ u
  >>   show x ∈ s ∩ u,
⊢ x ∈ s ∩ u
  >>   exact ⟨xs, xu⟩ },
no goals
-/

-- 2ª demostración
-- ===============

example : s ∩ (t ∪ u) ⊆ (s ∩ t) ∪ (s ∩ u) :=
begin
  rintros x ⟨xs, xt | xu⟩,
  { left,
    exact ⟨xs, xt⟩ },
  { right,
    exact ⟨xs, xu⟩ },
end

-- Prueba
-- ======

/-
α : Type u_1,
s t u : set α
⊢ s ∩ (t ∪ u) ⊆ s ∩ t ∪ s ∩ u
  >> rintros x ⟨xs, xt | xu⟩,
| x : α,
```

```
| xs : x ∈ s,
| xt : x ∈ t
| ⊢ x ∈ s ∩ t ∪ s ∩ u
|   >> { left,
| ⊢ x ∈ s ∩ t
|   >>   exact ⟨xs, xt⟩ },
xu : x ∈ u
⊢ x ∈ s ∩ t ∪ s ∩ u
  >> { right,
⊢ x ∈ s ∩ u
  >>   exact ⟨xs, xu⟩ },
no goals
-/

-- 3ª demostración
-- ===============

example : s ∩ (t ∪ u) ⊆ (s ∩ t) ∪ (s ∩ u) :=
begin
 rw set.inter_distrib_left,
end

-- ----------------------------------------------------------------------
-- Ejercicio. Demostrar que
--    (s ∩ t) ∪ (s ∩ u) ⊆ s ∩ (t ∪ u)
-- ----------------------------------------------------------------------


example : (s ∩ t) ∪ (s ∩ u) ⊆ s ∩ (t ∪ u) :=
begin
  rintros x (⟨xs,xt⟩ | ⟨xs,xu⟩),
  { split,
    { exact xs },
    { left,
      exact xt }},
  { split,
    { exact xs },
    { right,
      exact xu }},
end

-- Prueba
-- ======

/-
```

```
α : Type u_1,
s t u : set α
⊢ (s ∩ t) ∪ (s ∩ u) ⊆ s ∩ (t ∪ u)
  >> rintros x (⟨xs,xt⟩ | ⟨xs,xu⟩),
| x : α,
| xs : x ∈ s,
| xt : x ∈ t
| ⊢ x ∈ s ∩ (t ∪ u)
|   >> { split,
| | ⊢ x ∈ s
| |   >>   { exact xs },
| | ⊢ x ∈ t ∪ u
| |   >>   { left,
| | ⊢ x ∈ t
| |   >>     exact xt }},
x : α,
xs : x ∈ s,
xu : x ∈ u
⊢ x ∈ s ∩ (t ∪ u)
  >> { split,
| ⊢ x ∈ s
|   >>   { exact xs },
⊢ x ∈ t ∪ u
  >>   { right,
⊢ x ∈ u
  >>     exact xu }},
no goals
-/
```

- Diferencia de diferencia

```
-- ---------------------------------------------------------------------
-- Ejercicio. Demostrar que
--    (s \ t) \ u ⊆ s \ (t ∪ u)
-- ---------------------------------------------------------------------

import tactic

variable {α : Type*}
variables (s t u : set α)

-- 1ª demostración
-- ===============
```

```
example : (s \ t) \ u ⊆ s \ (t ∪ u) :=
begin
  intros x xstu,
  have xs : x ∈ s := xstu.1.1,
  have xnt : x ∉ t := xstu.1.2,
  have xnu : x ∉ u := xstu.2,
  split,
  { exact xs },
  { dsimp,
    intro xtu,
    cases xtu with xt xu,
    { show false,
      from xnt xt },
    { show false,
      from xnu xu }},
end

-- Prueba
-- ======

/-
α : Type u_1,
s t u : set α
⊢ s \ t \ u ⊆ s \ (t ∪ u)
  >> intros x xstu,
x : α,
xstu : x ∈ s \ t \ u
⊢ x ∈ s \ (t ∪ u)
  >> have xs : x ∈ s := xstu.1.1,
xs : x ∈ s
⊢ x ∈ s \ (t ∪ u)
  >> have xnt : x ∉ t := xstu.1.2,
xnt : x ∉ t
⊢ x ∈ s \ (t ∪ u)
  >> have xnu : x ∉ u := xstu.2,
xnu : x ∉ u
⊢ x ∈ s \ (t ∪ u)
  >> split,
| ⊢ x ∈ s
|   >> { exact xs },
⊢ (λ (a : α), a ∉ t ∪ u) x
  >> { dsimp,
⊢ ¬(x ∈ t ∨ x ∈ u)
  >>   intro xtu,
```

```
xtu : x ∈ t ∨ x ∈ u
⊢ false
  >>   cases xtu with xt xu,
| xt : x ∈ t
| ⊢ false
|   >>   { show false,
| ⊢ false
|   >>     from xnt xt },
xu : x ∈ u
⊢ false
  >>   { show false,
⊢ false
  >>     from xnu xu }},
no goals
-/

-- 2ª demostración
-- ===============

example : s \ t \ u ⊆ s \ (t ∪ u) :=
begin
  rintros x ⟨⟨xs, xnt⟩, xnu⟩,
  use xs,
  rintros (xt | xu),
  { contradiction },
  { contradiction },
end

-- Prueba
-- ======

/-
α : Type u_1,
s t u : set α
⊢ s \ t \ u ⊆ s \ (t ∪ u)
  >> rintros x ⟨⟨xs, xnt⟩, xnu⟩,
x : α,
xnu : x ∉ u,
xs : x ∈ s,
xnt : x ∉ t
⊢ x ∈ s \ (t ∪ u)
  >> use xs,
⊢ (λ (a : α), a ∉ t ∪ u) x
  >> rintros (xt | xu),
| xt : x ∈ t
```

```
| ⊢ false
|   >> { contradiction },
xu : x ∈ u
⊢ false
  >> { contradiction },
no goals
-/



-- 3ª demostración
-- ===============

example : s \ t \ u ⊆ s \ (t ∪ u) :=
begin
  rintros x ⟨⟨xs, xnt⟩, xnu⟩,
  use xs,
  rintros (xt | xu); contradiction
end

-- ---------------------------------------------------------------------
-- Ejercicio. Demostrar que
--    s \ (t ∪ u) ⊆ (s \ t) \ u
-- ---------------------------------------------------------------------

example : s \ (t ∪ u) ⊆ (s \ t) \ u :=
begin
  rintros x ⟨xs, xntu⟩,
  use xs,
  { intro xt,
    exact xntu (or.inl xt) },
  { intro xu,
    apply xntu (or.inr xu) },
end

-- Prueba
-- ======

/-
α : Type u_1,
s t u : set α
⊢ s \ (t ∪ u) ⊆ s \ t \ u
  >> rintros x ⟨xs, xntu⟩,
x : α,
xs : x ∈ s,
```

```
xntu : x ∉ t ∪ u
⊢ x ∈ s \ t \ u
  >> use xs,
| ⊢ (λ (a : α), a ∉ t) x
|   >> { intro xt,
| xt : x ∈ t
| ⊢ false
|   >>   exact xntu (or.inl xt) },
⊢ (λ (a : α), a ∉ u) x
  >> { intro xu,
xu : x ∈ u
⊢ false
  >>   apply xntu (or.inr xu) },
no goals
-/
```

- Conmutativa de la intersección

```
-- ---------------------------------------------------------------------
-- Ejercicio. Demostrar que
--    s ∩ t = t ∩ s
-- ---------------------------------------------------------------------

import tactic

open set

variable {α : Type*}
variables (s t u : set α)

-- 1ª demostración
-- ===============

example : s ∩ t = t ∩ s :=
begin
  ext x,
  simp only [mem_inter_eq],
  split,
  { rintros ⟨xs, xt⟩,
    exact ⟨xt, xs⟩ },
  { rintros ⟨xt, xs⟩,
    exact ⟨xs, xt⟩ },
end
```

```
-- Prueba
-- ======

/-
α : Type u_1,
s t : set α
⊢ s ∩ t = t ∩ s
  >> ext x,
x : α
⊢ x ∈ s ∩ t ↔ x ∈ t ∩ s
  >> simp only [mem_inter_eq],
⊢ x ∈ s ∧ x ∈ t ↔ x ∈ t ∧ x ∈ s
  >> split,
| ⊢ x ∈ s ∧ x ∈ t → x ∈ t ∧ x ∈ s
|   >> { rintros ⟨xs, xt⟩,
| xs : x ∈ s,
| xt : x ∈ t
| ⊢ x ∈ t ∧ x ∈ s
|   >>   exact ⟨xt, xs⟩ },
⊢ x ∈ t ∧ x ∈ s → x ∈ s ∧ x ∈ t
  >> { rintros ⟨xt, xs⟩,
xt : x ∈ t,
xs : x ∈ s
⊢ x ∈ s ∧ x ∈ t
  >>   exact ⟨xs, xt⟩ },
no goals
-/

-- Comentarios:
-- 1. La táctica (ext x) transforma la conclusión (s = t) en
--    (x ∈ s ↔ x ∈ t).
-- 2. Se ha usado el lema
--    + mem_inter_eq x s t : x ∈ s ∩ t = (x ∈ s ∧ x ∈ t)

-- Comprobación:
variable (x : α)
#check @mem_inter_eq _ x s t

-- 2ª demostración
-- ===============

example : s ∩ t = t ∩ s :=
ext $ λ x, ⟨λ ⟨xs, xt⟩, ⟨xt, xs⟩, λ ⟨xt, xs⟩, ⟨xs, xt⟩⟩

-- Comentario: La notación `f $ ...`  es equivalente a `f (...)`.
```

```
-- 3ª demostración
-- ===============

example : s ∩ t = t ∩ s :=
by ext x; simp [and.comm]

-- 4ª demostración
-- ===============

example : s ∩ t = t ∩ s :=
inf_comm

-- 5ª demostración
-- ===============

example : s ∩ t = t ∩ s :=
begin
  apply subset.antisymm,
  { rintros x ⟨xs, xt⟩,
    exact ⟨xt, xs⟩ },
  { rintros x ⟨xt, xs⟩,
    exact ⟨xs, xt⟩ },
end

-- Prueba
-- ======

/-
α : Type u_1,
s t : set α
⊢ s ∩ t = t ∩ s
  >> apply subset.antisymm,
| ⊢ s ∩ t ⊆ t ∩ s
|   >> { rintros x ⟨xs, xt⟩,
| x : α,
| xs : x ∈ s,
| xt : x ∈ t
| ⊢ x ∈ t ∩ s
|   >>   exact ⟨xt, xs⟩ },
⊢ t ∩ s ⊆ s ∩ t
  >> { rintros x ⟨xt, xs⟩,
x : α,
xt : x ∈ t,
xs : x ∈ s
```

```
⊢ x ∈ s ∩ t
  >>   exact ⟨xs, xt⟩ },
no goals
-/
```

- Identidades conjuntistas

```
-- ---------------------------------------------------------------------
-- Ejercicio. Demostrar que
--    s ∩ (s ∪ t) = s
-- ---------------------------------------------------------------------

import tactic

open set

variable {α : Type*}
variables (s t u : set α)

example : s ∩ (s ∪ t) = s :=
begin
  apply subset.antisymm,
  { rintros x ⟨xs,_⟩,
    exact xs },
  { rintros x xs,
    split,
    { exact xs },
    { apply mem_union_left,
      exact xs }},
end

-- Prueba
-- ======

/-
α : Type u_1,
s t : set α
⊢ s ∩ (s ∪ t) = s
  >> apply subset.antisymm,
| ⊢ s ∩ (s ∪ t) ⊆ s
|   >> { rintros x ⟨xs,_⟩,
| x : α,
| xs : x ∈ s,
| a_right : x ∈ s ∪ t
```

```
| ⊢ x ∈ s
|   >>   exact xs },
⊢ s ⊆ s ∩ (s ∪ t)
  >> { rintros x xs
x : α,
xs : x ∈ s
⊢ x ∈ s ∩ (s ∪ t),
  >>   split,
| ⊢ x ∈ s
|   >>   { exact xs },
⊢ x ∈ s ∪ t
  >>   { apply mem_union_left,
⊢ x ∈ s
  >>     exact xs }},
no goals
-/

-- ---------------------------------------------------------------------
-- Ejercicio. Demostrar que
--    s ∪ (s ∩ t) = s
-- ---------------------------------------------------------------------

example : s ∪ (s ∩ t) = s :=
begin
  apply subset.antisymm,
  { rintros x (xs | ⟨xs,xt⟩),
    { exact xs },
    { exact xs }},
  { rintros x xs,
    left,
    exact xs },
end

-- Prueba
-- ======

/-
α : Type u_1,
s t : set α
⊢ s ∪ s ∩ t = s
  >> apply subset.antisymm,
| ⊢ s ∪ s ∩ t ⊆ s
|   >> { rintros x (xs | ⟨xs,xt⟩),
| | x : α,
| | xs : x ∈ s
```

```
| | ⊢ x ∈ s
| |  >>   { exact xs },
| x : α,
| xs : x ∈ s,
| xt : x ∈ t
| ⊢ x ∈ s
|   >>   { exact xs }},
⊢ s ⊆ s ∪ s ∩ t
  >> { rintros x xs,
x : α,
xs : x ∈ s
⊢ x ∈ s ∪ s ∩ t
  >>   left,
⊢ x ∈ s
  >>   exact xs },
no goals
-/

-- ---------------------------------------------------------------------
-- Ejercicio. Demostrar que
--    (s \ t) ∪ t = s ∪ t
-- ---------------------------------------------------------------------


example : (s \ t) ∪ t = s ∪ t :=
begin
  ext x,
  split,
  { rintros (⟨xs, xnt⟩ | xt),
    { left,
      exact xs },
    { right,
    exact xt }},
  rintros (xs | xt),
  { classical,
    by_cases xt : x ∈ t,
    { right,
      exact xt },
    { left,
      exact ⟨xs, xt⟩ }},
  { right,
    exact xt },

end
```

```
-- ---------------------------------------------------------------------
-- Ejercicio. Demostrar que
--    (s \ t) ∪ (t \ s) = (s ∪ t) \ (s ∩ t)
-- ---------------------------------------------------------------------

example : (s \ t) ∪ (t \ s) = (s ∪ t) \ (s ∩ t) :=
begin
  apply  subset.antisymm,
  { rintros x (⟨xs,xnt⟩ | ⟨xt,xns⟩),
    { split,
      { exact mem_union_left t xs },
      { intro h,
        apply xnt,
        exact mem_of_mem_inter_right h }},
    { split,
      { exact mem_union_right s xt },
      { intro h,
        apply xns,
        exact mem_of_mem_inter_left h }}},
  { rintros x ⟨xs | xt,xnst⟩,
    { left,
      split,
      { exact xs },
      { intro h,
        apply xnst,
        exact mem_sep xs h }},
    { right,
      split,
      { exact xt },
      { intro h,
        apply xnst,
        exact mem_sep h xt }}},
end

-- Prueba
-- ======

/-
α : Type u_1,
s t : set α
⊢ s \ t ∪ t \ s = (s ∪ t) \ (s ∩ t)
  apply  subset.antisymm,
| ⊢ s \ t ∪ t \ s ⊆ (s ∪ t) \ (s ∩ t)
|   { rintros x (⟨xs,xnt⟩ | ⟨xt,xns⟩)},
| | x : α,
```

```
| | xs : x ∈ s,
| | xnt : x ∉ t
| | ⊢ x ∈ (s ∪ t) \ (s ∩ t)
| |     { split,
| | | ⊢ x ∈ s ∪ t
| | |       { exact mem_union_left t xs },
| | ⊢ (λ (a : α), a ∉ s ∩ t) x
| |       { intro h,
| | h : x ∈ s ∩ t
| | ⊢ false
| |         apply xnt,
| | ⊢ x ∈ t
| |         exact mem_of_mem_inter_right h }},
| x : α,
| xt : x ∈ t,
| xns : x ∉ s
| ⊢ x ∈ (s ∪ t) \ (s ∩ t)
|     { split,
| | ⊢ x ∈ s ∪ t
| |       { exact mem_union_right s xt },
| ⊢ (λ (a : α), a ∉ s ∩ t) x
|       { intro h,
| h : x ∈ s ∩ t
| ⊢ false
|         apply xns,
| ⊢ x ∈ s
|         exact mem_of_mem_inter_left h }}},
⊢ (s ∪ t) \ (s ∩ t) ⊆ s \ t ∪ t \ s
  { rintros x ⟨xs | xt,xnst⟩,
| x : α,
| xnst : x ∉ s ∩ t,
| xs : x ∈ s
| ⊢ x ∈ s \ t ∪ t \ s
|     { left,
| ⊢ x ∈ s \ t
|       split,
| ⊢ x ∈ s
| |       { exact xs },
| ⊢ (λ (a : α), a ∉ t) x
|       { intro h,
| h : x ∈ t
| ⊢ false
|         apply xnst,
| ⊢ x ∈ s ∩ t
|         exact mem_sep xs h }},
```

```
⊢ x ∈ s \ t ∪ t \ s
    { right,
⊢ x ∈ t \ s
      split,
| ⊢ x ∈ t
|       { exact xt },
⊢ (λ (a : α), a ∉ s) x
      { intro h,
h : x ∈ s
⊢ false
        apply xnst,
⊢ x ∈ s ∩ t
        exact mem_sep h xt }}},
no goals
-/


#lint
```

- Unión de pares e impares

```
-- ---------------------------------------------------------------------
-- Ejercicio. Ejercutar las siguientes acciones
-- 1. Importar la librería data.set.basic data.nat.parity
-- 2. Abrir los espacios de nombres set y nat.
-- ---------------------------------------------------------------------

import data.set.basic data.nat.parity

open set nat

-- ---------------------------------------------------------------------
-- Ejercicio. Definir el conjunto de los números pares.
-- ---------------------------------------------------------------------

def evens : set ℕ := {n | even n}

-- ---------------------------------------------------------------------
-- Ejercicio. Definir el conjunto de los números impares.
-- ---------------------------------------------------------------------

def odds :  set ℕ := {n | ¬ even n}

-- ---------------------------------------------------------------------
```

```
-- Ejercicio. Demostrar la unión de los pares e impares es el universal.
-- ----------------------------------------------------------------------

-- 1ª demostración
-- ===============

example : evens ∪ odds = univ :=
begin
  rw [evens, odds],
  ext n,
  simp,
  apply classical.em,
end

-- Prueba
-- ======

/-
⊢ evens ∪ odds = univ
  >> rw [evens, odds],
⊢ {n : ℕ | n.even} ∪ {n : ℕ | ¬n.even} = univ
  >> ext n,
n : ℕ
⊢ n ∈ {n : ℕ | n.even} ∪ {n : ℕ | ¬n.even} ↔ n ∈ univ
  >> simp,
⊢ n.even ∨ ¬n.even
  >> apply classical.em,
no goals
-/

-- 2ª demostración
-- ===============

example : evens ∪ odds = univ :=
begin
  ext n,
  simp,
  apply classical.em,
end

-- Prueba
-- ======

/-
⊢ evens ∪ odds = univ
```

```
  >> ext n,
n : ℕ
⊢ n ∈ evens ∪ odds ↔ n ∈ univ
  >> simp,
⊢ n ∈ evens ∨ n ∈ odds
  >> apply classical.em,
no goals
-/
```

- Pertenencia al vacío y al universal

```
-- ---------------------------------------------------------------------
-- Ejercicio. Abrir el espacion de nombres set.
-- ---------------------------------------------------------------------

import tactic

open set

-- ---------------------------------------------------------------------
-- Ejercicio. Demostrar ningún elemento pertenece al vacío.
-- ---------------------------------------------------------------------

example (x : ℕ) : x ∉ (∅ : set ℕ) :=
not_false

example (x : ℕ) (h : x ∈ (∅ : set ℕ)) : false :=
h

-- ---------------------------------------------------------------------
-- Ejercicio. Demostrar tofos los elementos pertenecen al universal.
-- ---------------------------------------------------------------------

example (x : ℕ) : x ∈ (univ : set ℕ) :=
trivial
```

- Primos mayores que dos

```
-- ---------------------------------------------------------------------
-- Ejercicio. Demostrar que los primos mayores que 2 son impares.
-- ---------------------------------------------------------------------

import data.nat.prime data.nat.parity tactic
```

```
open set nat

example : { n | prime n } ∩ { n | n > 2} ⊆ { n | ¬ even n } :=
begin
  intro n,
  simp,
  intro nprime,
  cases prime.eq_two_or_odd nprime with h h,
  { rw h,
    intro,
    linarith },
  { rw even_iff,
    rw h,
    norm_num },
end

-- Prueba
-- ======

/-
⊢ {n : ℕ | n.prime} ∩ {n : ℕ | n > 2} ⊆ {n : ℕ | ¬n.even}
  >> intro n,
n : ℕ
⊢ n ∈ {n : ℕ | n.prime} ∩ {n : ℕ | n > 2} → n ∈ {n : ℕ | ¬n.even}
  >> simp,
⊢ n.prime → 2 < n → ¬n.even
  >> intro nprime,
nprime : n.prime
⊢ 2 < n → ¬n.even
  >> cases prime.eq_two_or_odd nprime with h h,
| h : n = 2
| ⊢ 2 < n → ¬n.even
|   >> { rw h,
| ⊢ 2 < 2 → ¬2.even
|   >>   intro,
| a : 2 < 2
| ⊢ ¬2.even
|   >>   linarith },
h : n % 2 = 1
⊢ 2 < n → ¬n.even
  >> { rw even_iff,
⊢ 2 < n → ¬n % 2 = 0
  >>   rw h,
⊢ 2 < n → ¬1 = 0
```

```
  >>   norm_num },
no goals
-/

-- Comentario: Se han usado los lemas
-- + prime.eq_two_or_odd : p.prime → p = 2 ∨ p % 2 = 1
-- + even_iff : even n ↔ n % 2 = 0

variables (n p : ℕ)
#check @prime.eq_two_or_odd p
#check @even_iff n
```

- Ejemplos con cuantificadores acotados

```
-- ---------------------------------------------------------------------
-- Ejercicio. Realizar las siguientes acciones:
-- 1. Importar la librería data.nat.prime data.nat.parity
-- 2. Abrir el espacio de nombres nat.
-- 3. Declarar s como variable sobre conjuntos de números naturales.
-- ---------------------------------------------------------------------

import data.nat.prime data.nat.parity   -- 1
open nat                                -- 2
variable (s : set ℕ)                    -- 3

-- ---------------------------------------------------------------------
-- Ejercicio. Demostrar que si los elementos de s no son pares y si los
-- elementos de s son primos, entonces los elementos de s no son pares
-- pero sí son primpos.
-- ---------------------------------------------------------------------

example
  (h₀ : ∀ x ∈ s, ¬ even x)
  (h₁ : ∀ x ∈ s, prime x)
  : ∀ x ∈ s, ¬ even x ∧ prime x :=
begin
  intros x xs,
  split,
  { apply h₀ x xs },
  { apply h₁ x xs },
end

-- Prueba
-- ======
```

```
/-
s : set ℕ,
h₀ : ∀ (x : ℕ), x ∈ s → ¬x.even,
h₁ : ∀ (x : ℕ), x ∈ s → x.prime
⊢ ∀ (x : ℕ), x ∈ s → ¬x.even ∧ x.prime
  >> intros x xs,
x : ℕ,
xs : x ∈ s
⊢ ¬x.even ∧ x.prime
  >> split,
| ⊢ ¬x.even
|   >> { apply h₀ x xs },
⊢ x.prime
  >> { apply h₁ x xs },
no goals
-/

-- Comentario: La táctica (intros x xs) si la conclusión es (∀ y ∈ s, P y)
-- y una hipótesis es (s : set α), entonces añade las hipótesis (x : α)
-- y (xs : x ∈ s) y cambia la conclusión a (P x).

-- ---------------------------------------------------------------------
-- Ejercicio. Demostrar que si s tiene algún elemento primo impar,
-- entonces tiene algún elemento primo.
-- ---------------------------------------------------------------------

example
  (h : ∃ x ∈ s, ¬ even x ∧ prime x)
  : ∃ x ∈ s, prime x :=
begin
  rcases h with ⟨x, xs, _, prime_x⟩,
  use [x, xs, prime_x],
end

-- Prueba
-- ======

/-
s : set ℕ,
h : ∃ (x : ℕ) (H : x ∈ s), ¬x.even ∧ x.prime
⊢ ∃ (x : ℕ) (H : x ∈ s), x.prime
  >> rcases h with ⟨x, xs, _, prime_x⟩,
xs : x ∈ s,
h_h_h_left : ¬x.even,
```

```
prime_x : x.prime
⊢ ∃ (x : ℕ) (H : x ∈ s), x.prime
  >> use [x, xs, prime_x],
no goals
-/

-- Comentarios:
-- 1. La táctica (cases h with ⟨x, xs, h1, h2⟩) si la
--    hipótesis es (∃ y ∈ s, P y ∧ Q y) y una hipótesis es (s : set α),
--    entonces quita h y añade las hipótesis (x : s), (h1 : P x) y
--    (h2 : Q x).
-- 2. La táctica (use [x, xs, h]) resuelve la conclusión
--    (∃ x ∈ s, P x) si xs es una prueba de (x ∈ s) y h lo es de (P x).
```

■ Ejercicios con cuantificadores acotados

```
-- ---------------------------------------------------------------------
-- Ejercicio. Realizar las siguientes acciones:
-- 1. Importar la librería data.nat.prime data.nat.parity
-- 2. Abrir el espacio de nombres nat
-- 3. Declarar s y t como variables sobre conjuntos de números naturales.
-- 4. Declarar el hecho (ssubt : s ⊆ t)
-- 5, Añadir ssubt como hipótesis de la teoría.
-- ---------------------------------------------------------------------

import data.nat.prime data.nat.parity  -- 1
open nat                               -- 2
variables (s t : set ℕ)                -- 3
variables (ssubt : s ⊆ t)              -- 4
include ssubt                          -- 5

-- ---------------------------------------------------------------------
-- Ejercicio. Demostrar que si
--    ∀ x ∈ t, ¬ even x
--    ∀ x ∈ t, prime x
-- entonces
--    ∀ x ∈ s, ¬ even x ∧ prime x
-- ---------------------------------------------------------------------

example
  (h₀ : ∀ x ∈ t, ¬ even x)
  (h₁ : ∀ x ∈ t, prime x)
  : ∀ x ∈ s, ¬ even x ∧ prime x :=
begin
```

```
  intros x xs,
  split,
  { apply h₀ x (ssubt xs) },
  { apply h₁ x (ssubt xs) },
end

-- Prueba
-- ======

/-
s t : set ℕ,
ssubt : s ⊆ t,
h₀ : ∀ (x : ℕ), x ∈ t → ¬x.even,
h₁ : ∀ (x : ℕ), x ∈ t → x.prime
⊢ ∀ (x : ℕ), x ∈ s → ¬x.even ∧ x.prime
  >> intros x xs,
x : ℕ,
xs : x ∈ s
⊢ ¬x.even ∧ x.prime
  >> split,
| ⊢ ¬x.even
|   >> { apply h₀ x (ssubt xs) },
⊢ x.prime
  >> { apply h₁ x (ssubt xs) },
no goals
-/

-- ---------------------------------------------------------------------
-- Ejercicio. Demostrar que si
--    ∃ x ∈ s, ¬ even x ∧ prime x
-- entonces
--   ∃ x ∈ t, prime x
-- ---------------------------------------------------------------------

example
  (h : ∃ x ∈ s, ¬ even x ∧ prime x)
  : ∃ x ∈ t, prime x :=
begin
  rcases h with ⟨x, xs, _, px⟩,
  use [x, ssubt xs, px],
end

-- Prueba
-- ======
```

```
/-
s t : set ℕ,
ssubt : s ⊆ t,
h : ∃ (x : ℕ) (H : x ∈ s), ¬x.even ∧ x.prime
⊢ ∃ (x : ℕ) (H : x ∈ t), x.prime
  >> rcases h with ⟨x, xs, _, px⟩,
x : ℕ,
xs : x ∈ s,
h_h_h_left : ¬x.even,
px : x.prime
⊢ ∃ (x : ℕ) (H : x ∈ t), x.prime
  >> use [x, ssubt xs, px],
no goals
-/
```

- Ejemplos de uniones e intersecciones generales

```
-- ---------------------------------------------------------------------
-- Ejercicio. Realizar las siguientes acciones
-- 1. Importar la librería tactic
-- 2. Abrir el espacio de nombres set
-- 3. Declarar u y v como variables de universos.
-- 4. Declarar α como una variable de tipos en u.
-- 5. Declarar I como una variable de tipos en v.
-- 6. Declarar A y B como variables sobre funciones de I en α.
-- 7. Declarar s como variable sobre conjuntos de elementos de α.
-- ---------------------------------------------------------------------

import tactic                 -- 1
open set                      -- 2
universes u v                 -- 3
variable (α : Type u)         -- 4
variable (I : Type v)         -- 5
variables (A B : I → set α)   -- 6
variable  s : set α           -- 7

-- ---------------------------------------------------------------------
-- Ejercicio. Demostrar que
--    s ∩ (⋃ i, A i) = ⋃ i, (A i ∩ s)
-- ---------------------------------------------------------------------

example : s ∩ (⋃ i, A i) = ⋃ i, (A i ∩ s) :=
begin
  ext x,
```

```
  simp only [mem_inter_eq, mem_Union],
  split,
  { rintros ⟨xs, ⟨i, xAi⟩⟩,
    exact ⟨i, xAi, xs⟩ },
  { rintros ⟨i, xAi, xs⟩,
    exact ⟨xs, ⟨i, xAi⟩⟩ },
end

-- Prueba
-- ======

/-
α : Type u,
I : Type v,
A : I → set α,
s : set α
⊢ (s ∩ ⋃ (i : I), A i) = ⋃ (i : I), A i ∩ s
  >> ext x,
x : α
⊢ (x ∈ s ∩ ⋃ (i : I), A i) ↔ x ∈ ⋃ (i : I), A i ∩ s
  >> simp only [mem_inter_eq, mem_Union],
⊢ (x ∈ s ∧ ∃ (i : I), x ∈ A i) ↔ ∃ (i : I), x ∈ A i ∧ x ∈ s
  >> split,
| ⊢ (x ∈ s ∧ ∃ (i : I), x ∈ A i) → (∃ (i : I), x ∈ A i ∧ x ∈ s)
|   >> { rintros ⟨xs, ⟨i, xAi⟩⟩,
| x : α,
| xs : x ∈ s,
| i : I,
| xAi : x ∈ A i
| ⊢ ∃ (i : I), x ∈ A i ∧ x ∈ s
|   >>   exact ⟨i, xAi, xs⟩ },
⊢ (∃ (i : I), x ∈ A i ∧ x ∈ s) → (x ∈ s ∧ ∃ (i : I), x ∈ A i)
  >> { rintros ⟨i, xAi, xs⟩,
x : α,
i : I,
xAi : x ∈ A i,
xs : x ∈ s
⊢ x ∈ s ∧ ∃ (i : I), x ∈ A i
  >>   exact ⟨xs, ⟨i, xAi⟩⟩ },
no goals
-/

-- Comentario: Se han usado los lemas
-- + mem_inter_eq: x ∈ a ∩ b = (x ∈ a ∧ x ∈ b)
-- + mem_Union : x ∈ Union A ↔ ∃ (i : I), x ∈ A i
```

```
-- Comprobación
variable x : α
variables (a b : set α)
#check @mem_inter_eq _ x a b
#check @mem_Union α I x A

-- ---------------------------------------------------------------------
-- Ejercicio. Demostrar que
--    (⋂ i, A i ∩ B i) = (⋂ i, A i) ∩ (⋂ i, B i)
-- ---------------------------------------------------------------------

example : (⋂ i, A i ∩ B i) = (⋂ i, A i) ∩ (⋂ i, B i) :=
begin
  ext x,
  simp only [mem_inter_eq, mem_Inter],
  split,
  { intro h,
    split,
    { intro i,
      exact (h i).1 },
    { intro i,
      exact (h i).2 }},
  { rintros ⟨h1, h2⟩ i,
    split,
    { exact h1 i },
    { exact h2 i }},
end

-- Prueba
-- ======

/-
α : Type u,
I : Type v,
A B : I → set α
⊢ (⋂ (i : I), A i ∩ B i) = (⋂ (i : I), A i) ∩ ⋂ (i : I), B i
  >> ext x,
x : α
⊢ (x ∈ ⋂ (i : I), A i ∩ B i) ↔ x ∈ (⋂ (i : I), A i) ∩ ⋂ (i : I), B i
  >> simp only [mem_inter_eq, mem_Inter],
⊢ (∀ (i : I), x ∈ A i ∧ x ∈ B i) ↔ (∀ (i : I), x ∈ A i) ∧ ∀ (i : I), x ∈ B i
  >> split,
| ⊢ (∀ (i : I), x ∈ A i ∧ x ∈ B i) → ((∀ (i : I), x ∈ A i) ∧ ∀ (i : I), x ∈ B i)
|   >> { intro h,
```

```
| h : ∀ (i : I), x ∈ A i ∧ x ∈ B i
| ⊢ (∀ (i : I), x ∈ A i) ∧ ∀ (i : I), x ∈ B i
|   >>   split,
| | ⊢ ∀ (i : I), x ∈ A i
| |   >>   { intro i,
| | i : I
| | ⊢ x ∈ A i
| |   >>     exact (h i).1 },
| ⊢ ∀ (i : I), x ∈ B i
|   >>   { intro i,
| i : I
| ⊢ x ∈ B i
|   >>     exact (h i).2 }},
⊢ ((∀ (i : I), x ∈ A i) ∧ ∀ (i : I), x ∈ B i) → ∀ (i : I), x ∈ A i ∧ x ∈ B i
  >> { rintros ⟨h1, h2⟩ i,
i : I,
h1 : ∀ (i : I), x ∈ A i,
h2 : ∀ (i : I), x ∈ B i
⊢ x ∈ A i ∧ x ∈ B i
  >>   split,
| ⊢ x ∈ A i
|   >>   { exact h1 i },
⊢ x ∈ B i
  >>   { exact h2 i }},
no goals
-/

-- Comentario: Se han usado los lemas
-- + mem_inter_eq: x ∈ a ∩ b = (x ∈ a ∧ x ∈ b)
-- + mem_Inter : x ∈ Inter A ↔ ∀ (i : I), x ∈ A i

-- Comprobación
#check @mem_inter_eq _ x a b
#check @mem_Inter α I x A
```

- Ejercicios de uniones e intersecciones generales

```
-- ----------------------------------------------------------------------
-- Ejercicio. Realizar las siguientes acciones
-- 1. Importar la librería tactic
-- 2. Abrir el espacio de nombres set
-- 3. Declarar u y v como variables de universos.
-- 4. Declarar α como una variable de tipos en u.
-- 5. Declarar I como una variable de tipos en v.
```

```
-- 6. Declarar A y B como variables sobre funciones de I en α.
-- 7. Declarar s como variable sobre conjuntos de elementos de α.
-- 8. Usar la lógica clásica.
-- ---------------------------------------------------------------------

import tactic                  -- 1
open set                       -- 2
universes u v                  -- 3
variable (α : Type u)          -- 4
variable (I : Type v)          -- 5
variables (A B : I → set α)    -- 6
variable  s : set α            -- 7
open_locale classical          -- 8

-- ---------------------------------------------------------------------
-- Ejercicio. Demostrar que
--    s ∪ (⋂ i, A i) = ⋂ i, (A i ∪ s)
-- ---------------------------------------------------------------------

example : s ∪ (⋂ i, A i) = ⋂ i, (A i ∪ s) :=
begin
  ext x,
  simp only [mem_union, mem_Inter],
  split,
  { rintros (xs | xI),
    { intro i,
      right,
      exact xs },
    { intro i,
      left,
      exact xI i }},
  { intro h,
    by_cases xs : x ∈ s,
    { left,
      exact xs },
    { right,
      intro i,
      cases h i,
      { assumption },
      { contradiction }}},
end

-- Prueba
-- ======
```

```
/-
α : Type u,
I : Type v,
A : I → set α,
s : set α
⊢ (s ∪ ⋂ (i : I), A i) = ⋂ (i : I), A i ∪ s
  >> ext x,
x : α
⊢ (x ∈ s ∪ ⋂ (i : I), A i) ↔ x ∈ ⋂ (i : I), A i ∪ s
  >> simp only [mem_union, mem_Inter],
⊢ (x ∈ s ∨ ∀ (i : I), x ∈ A i) ↔ ∀ (i : I), x ∈ A i ∨ x ∈ s
  >> split,
| ⊢ (x ∈ s ∨ ∀ (i : I), x ∈ A i) → ∀ (i : I), x ∈ A i ∨ x ∈ s
|   >> { rintros (xs | xI),
| | xs : x ∈ s
| | ⊢ ∀ (i : I), x ∈ A i ∨ x ∈ s
| |   >>   { intro i,
| | i : I
| | ⊢ x ∈ A i ∨ x ∈ s
| |   >>     right,
| | ⊢ x ∈ s
| |   >>     exact xs },
| ⊢ ∀ (i : I), x ∈ A i ∨ x ∈ s
|   >>   { intro i,
| i : I
| ⊢ x ∈ A i ∨ x ∈ s
|   >>     left,
| ⊢ x ∈ A i
|   >>     exact xI i }},
⊢ (∀ (i : I), x ∈ A i ∨ x ∈ s) → (x ∈ s ∨ ∀ (i : I), x ∈ A i)
  >> { intro h,
h : ∀ (i : I), x ∈ A i ∨ x ∈ s
⊢ x ∈ s ∨ ∀ (i : I), x ∈ A i
  >>   by_cases xs : x ∈ s,
| xs : x ∈ s
| ⊢ x ∈ s ∨ ∀ (i : I), x ∈ A i
|   >>   { left,
| ⊢ x ∈ s
|   >>     exact xs },
xs : x ∉ s
⊢ x ∈ s ∨ ∀ (i : I), x ∈ A i
  >>   { right,
⊢ ∀ (i : I), x ∈ A i
  >>     intro i,
i : I
```

```
⊢ x ∈ A i
  >>     cases h i,
| h_1 : x ∈ A i
| ⊢ x ∈ A i
|   >>     { assumption },
⊢ x ∈ A i
  >>     { contradiction }}},
no goals
-/
```

- Ejemplos de uniones e intersecciones generales (2)

```
-- ---------------------------------------------------------------------
-- Ejercicio. Realizar las siguientes acciones:
-- 1. Importar la teoría data.set.lattice
-- 2. Importar la teoría data.nat.prime
-- 3. Abrir los espacios de nombre set y nat.
-- ---------------------------------------------------------------------

import data.set.lattice   -- 1
import data.nat.prime     -- 2
open set nat              -- 3

-- ---------------------------------------------------------------------
-- Ejercicio. Definir el conjunto de los números primos.
-- ---------------------------------------------------------------------

def primes : set ℕ := {x | prime x}

-- ---------------------------------------------------------------------
-- Ejercicio. Demostrar que
--    (⋃ p ∈ primes, {x | p^2 | x}) = {x | ∃ p ∈ primes, p^2 | x} :=
-- ---------------------------------------------------------------------

-- 1ª demostración
-- ===============

example : (⋃ p ∈ primes, {x | p^2 ∣ x}) = {x | ∃ p ∈ primes, p^2 ∣ x} :=
begin
  ext,
  rw mem_bUnion_iff,
  refl,
end
```

```
-- Prueba
-- ======

/-
⊢ (⋃ (p : ℕ) (H : p ∈ primes), {x : ℕ | p ^ 2 | x}) =
    {x : ℕ | ∃ (p : ℕ) (H : p ∈ primes), p ^ 2 | x}
  >> ext,
x : ℕ
⊢ (x ∈ ⋃ (p : ℕ) (H : p ∈ primes), {x : ℕ | p ^ 2 | x}) ↔
    x ∈ {x : ℕ | ∃ (p : ℕ) (H : p ∈ primes), p ^ 2 | x}
  >> rw mem_bUnion_iff,
⊢ (∃ (x_1 : ℕ) (H : x_1 ∈ primes), x ∈ {x : ℕ | x_1 ^ 2 | x}) ↔
    x ∈ {x : ℕ | ∃ (p : ℕ) (H : p ∈ primes), p ^ 2 | x}
  >> refl,
no goals
-/

-- Comentario: Se ha usado el lema
-- + mem_bUnion_iff : y ∈ (⋃ x ∈ s, t x) ↔ ∃ x ∈ s, y ∈ t x

-- Comprobación:
universes u v
variable α : Type u
variable β : Type v
variable s : set α
variable t : α → set β
variable y : β
#check @mem_bUnion_iff α β s t y

example : y ∈ (⋃ x ∈ s, t x) ↔ ∃ x ∈ s, y ∈ t x :=
mem_bUnion_iff

-- 2ª demostración
-- ===============

example : (⋃ p ∈ primes, {x | p^2 ∣ x}) = {x | ∃ p ∈ primes, p^2 ∣ x} :=
by { ext, rw mem_bUnion_iff, refl }

-- ---------------------------------------------------------------------
-- Ejercicio. Demostrar que
--    (⋃ p ∈ primes, {x | p^2 ∣ x}) = {x | ∃ p ∈ primes, p^2 ∣ x}
-- ---------------------------------------------------------------------

example : (⋃ p ∈ primes, {x | p^2 ∣ x}) = {x | ∃ p ∈ primes, p^2 ∣ x} :=
by { ext, simp }
```

```
-- ---------------------------------------------------------------------
-- Ejercicio. Demostrar que
--    (⋂ p ∈ primes, {x | ¬ p | x}) ⊆ {x | x < 2}
-- ---------------------------------------------------------------------

example : (⋂ p ∈ primes, {x | ¬ p | x}) ⊆ {x | x < 2} :=
begin
  intro x,
  contrapose!,
  simp,
  apply exists_prime_and_dvd,
end

-- Prueba
-- ======

/-
⊢ (⋂ (p : ℕ) (H : p ∈ primes), {x : ℕ | ¬p | x}) ⊆ {x : ℕ | x < 2}
  >> intro x,
x : ℕ
⊢ (x ∈ ⋂ (p : ℕ) (H : p ∈ primes), {x : ℕ | ¬p | x}) → x ∈ {x : ℕ | x < 2}
  >> contrapose!,
⊢ x ∉ {x : ℕ | x < 2} → (x ∉ ⋂ (p : ℕ) (H : p ∈ primes), {x : ℕ | ¬p | x})
  >> simp,
⊢ 2 ≤ x → (∃ (x_1 : ℕ), x_1 ∈ primes ∧ x_1 | x)
  >> apply exists_prime_and_dvd,
no goals
-/

-- Comentario: Se ha aplicado el lema
-- + exists_prime_and_dvd : 2 ≤ n → (∃ (p : ℕ), p.prime ∧ p | n)

variable n : ℕ
#check @exists_prime_and_dvd n

-- ---------------------------------------------------------------------
-- Ejercicio. Demostrar que
--    (⋃ p ∈ primes, {x | x ≤ p}) = univ
-- ---------------------------------------------------------------------

example : (⋃ p ∈ primes, {x | x ≤ p}) = univ :=
begin
  apply eq_univ_of_forall,
  intro x,
```

```
  simp,
  rcases exists_infinite_primes x with ⟨p, pge, primep⟩,
  use [p, primep, pge],
end

-- Prueba
-- ======

/-
⊢ (⋃ (p : ℕ) (H : p ∈ primes), {x : ℕ | x ≤ p}) = univ
  >> apply eq_univ_of_forall,
⊢ ∀ (x : ℕ), x ∈ ⋃ (p : ℕ) (H : p ∈ primes), {x : ℕ | x ≤ p}
  >> intro x,
x : ℕ
⊢ x ∈ ⋃ (p : ℕ) (H : p ∈ primes), {x : ℕ | x ≤ p}
  >> simp,
⊢ ∃ (i : ℕ), i ∈ primes ∧ x ≤ i
  >> rcases exists_infinite_primes x with ⟨p, pge, primep⟩,
x p : ℕ,
pge : x ≤ p,
primep : p.prime
⊢ ∃ (i : ℕ), i ∈ primes ∧ x ≤ i
  >> use [p, primep, pge],
no goals
-/

-- Comentario: Se han usado los lemas
-- + eq_univ_of_forall : (∀ x, x ∈ s) → s = univ
-- + exists_infinite_primes : ∀ (n : ℕ), ∃ (p : ℕ), n ≤ p ∧ p.prime

variable α : Type*
variable s : set α
#check @eq_univ_of_forall α s
#check exists_infinite_primes
```

- Ejemplos de uniones e intersecciones generales (3)

```
-- ----------------------------------------------------------------------
-- Ejercicio. Realizar las siguientes acciones:
-- 1. Importar la librería data.set.lattice
-- 2. Abrir el espacio de nombres set.
-- 3. Declarar α una variable de tipos.
-- 4. Declarar s una vabiable sobre conjuntos de conjuntos de elementos
--    de α.
```

```
-- ---------------------------------------------------------------------

import data.set.lattice      -- 1
open set                     -- 2
variable {α : Type*}         -- 3
variable (s : set (set α))   -- 4

-- ---------------------------------------------------------------------
-- Ejercicio. Demostrar que
--    ⋃₀ s = ⋃ t ∈ s, t
-- ---------------------------------------------------------------------

-- 1ª demostración
-- ===============

example : ⋃₀ s = ⋃ t ∈ s, t :=
begin
  ext x,
  rw mem_bUnion_iff,
  refl,
end

-- Prueba
-- ======

/-
α : Type u_1,
s : set (set α)
⊢ ⋃₀ s = ⋃ (t : set α) (H : t ∈ s), t
  >> ext x,
x : α
⊢ x ∈ ⋃₀ s ↔ x ∈ ⋃ (t : set α) (H : t ∈ s), t
  >> rw mem_bUnion_iff,
⊢ x ∈ ⋃₀ s ↔ ∃ (x_1 : set α) (H : x_1 ∈ s), x ∈ x_1
  >> refl,
no goals
-/

-- Comentario: Se ha usado el lema
-- + mem_bUnion_iff: y ∈ (⋃ x ∈ s, t x) ↔ ∃ x ∈ s, y ∈ t x

-- 2ª demostración
-- ===============

example : ⋃₀ s = ⋃ t ∈ s, t :=
```

```
sUnion_eq_bUnion

-- ---------------------------------------------------------------------
-- Ejercicio. Demostrar que
--    ⋂₀ s = ⋂ t ∈ s, t
-- ---------------------------------------------------------------------

-- 1ª demostración
-- ===============

example : ⋂₀ s = ⋂ t ∈ s, t :=
begin
  ext x,
  rw mem_bInter_iff,
  refl,
end

-- Prueba
-- ======

/-
α : Type u_1,
s : set (set α)
⊢ ⋂₀ s = ⋂ (t : set α) (H : t ∈ s), t
  >> ext x,
x : α
⊢ x ∈ ⋂₀ s ↔ x ∈ ⋂ (t : set α) (H : t ∈ s), t
  >> rw mem_bInter_iff,
⊢ x ∈ ⋂₀ s ↔ ∀ (x_1 : set α), x_1 ∈ s → x ∈ x_1
  >> refl,
no goals
-/

-- Comentario: Se ha usado el lema
-- + mem_bInter_iff : y ∈ (⋂ x ∈ s, t x) ↔ ∀ x ∈ s, y ∈ t x

-- 2ª demostración
-- ===============

example : ⋂₀ s = ⋂ t ∈ s, t :=
sInter_eq_bInter
```

4.2. Funciones

- Preimagen de la intersección

```
-- ---------------------------------------------------------------------
-- Ejercicio. Demostrar que
--    f ⁻¹' (u ∩ v) = f ⁻¹' u ∩ f ⁻¹' v
-- ---------------------------------------------------------------------

import data.set.function

universes u v
variable  α : Type u
variable  β : Type v
variable  f : α → β
variables u v : set β

example : f ⁻¹' (u ∩ v) = f ⁻¹' u ∩ f ⁻¹' v :=
by { ext, refl }
```

- Imagen de la unión

```
-- ---------------------------------------------------------------------
-- Ejercicio. Demostrar que
--    f '' (s ∪ t) = (f '' s) ∪ (f '' t)
-- ---------------------------------------------------------------------

import data.set.function

universes u v
variable  α : Type u
variable  β : Type v
variable  f : α → β
variables s t : set α

example : f '' (s ∪ t) = (f '' s) ∪ (f '' t) :=
begin
  ext y,
  split,
  { rintros ⟨x, xs | xt, rfl⟩,
    { left,
      use [x, xs] },
    { right,
```

```
      use [x, xt] }},
  { rintros (⟨x, xs, rfl⟩ | ⟨x, xt, rfl⟩),
    { use [x, or.inl xs] },
    { use [x, or.inr xt] }},
end

-- Prueba
-- ======

/-
α : Type u,
β : Type v,
f : α → β,
s t : set α
⊢ f '' (s ∪ t) = f '' s ∪ f '' t
  >> ext y,
y : β
⊢ y ∈ f '' (s ∪ t) ↔ y ∈ f '' s ∪ f '' t
  >> split,
| ⊢ y ∈ f '' (s ∪ t) → y ∈ f '' s ∪ f '' t
|   >> { rintros ⟨x, xs | xt, rfl⟩,
| | ⊢ f x ∈ f '' s ∪ f '' t
| |   >>   { left,
| | ⊢ f x ∈ f '' s
| |   >>     use [x, xs] },
| ⊢ f x ∈ f '' s ∪ f '' t
|   >>   { right,
| ⊢ f x ∈ f '' t
|   >>     use [x, xt] }},
⊢ y ∈ f '' s ∪ f '' t → y ∈ f '' (s ∪ t)
  >> { rintros ((⟨x, xs, rfl⟩ | ⟨x, xt, rfl⟩)),
| x : α,
| xs : x ∈ s
| ⊢ f x ∈ f '' (s ∪ t)
|   >>   { use [x, or.inl xs] },
xt : x ∈ t
⊢ f x ∈ f '' (s ∪ t)
  >>   { use [x, or.inr xt] }},
no goals
-/
```

- Primagen de imagen

```
-- ---------------------------------------------------------------------
-- Ejercicio. Demostrar que
--    s ⊆ f ⁻¹' (f '' s)
-- ---------------------------------------------------------------------

import data.set.function

universes u v
variable  α : Type u
variable  β : Type v
variable  f : α → β
variables s t : set α

example : s ⊆ f ⁻¹' (f '' s) :=
begin
  intros x xs,
  show f x ∈ f '' s,
  use [x, xs],
end

-- Prueba
-- ======

/-
α : Type u,
β : Type v,
f : α → β,
s : set α
⊢ s ⊆ f ⁻¹' (f '' s)
  >> intros x xs,
x : α,
xs : x ∈ s
⊢ x ∈ f ⁻¹' (f '' s)
  >> show f x ∈ f '' s,
⊢ f x ∈ f '' s
  >> use [x, xs],
no goals
-/

-- 2ª demostración
-- ===============

example : s ⊆ f ⁻¹' (f '' s) :=
begin
  intros x xs,
```

```
  show f x ∈ f '' s,
  apply set.mem_image_of_mem f xs,
end

-- Prueba
-- ======

/-
α : Type u,
β : Type v,
f : α → β,
s : set α
⊢ s ⊆ f ⁻¹' (f '' s)
  >> intros x xs,
x : α,
xs : x ∈ s
⊢ x ∈ f ⁻¹' (f '' s)
  >> show f x ∈ f '' s,
⊢ f x ∈ f '' s
  >> apply set.mem_image_of_mem f xs,
no goals
-/

-- Comentario: Se ha usado el lema
-- + set.mem_image_of_mem : x ∈ a → f x ∈ f '' a
```

▪ Inclusión de la imagen

```
-- ---------------------------------------------------------------------
-- Ejercicio. Demostrar que
--    f '' s ⊆ t ↔ s ⊆ f ⁻¹' t
-- ---------------------------------------------------------------------

import data.set.function

universes u v
variable  α : Type u
variable  β : Type v
variable  f : α → β
variables (s : set α) (t : set β)

open set

example : f '' s ⊆ t ↔ s ⊆ f ⁻¹' t :=
```

```
begin
  split,
  { intros h x xs,
    show f x ∈ t,
      { calc f x ∈ f '' s : mem_image_of_mem f xs
             ... ⊆ t      : h }},
  { intros h y hy,
    rcases hy with ⟨x,xs,fxy⟩,
    rw ← fxy,
    apply h,
    exact xs },
end

-- Prueba
-- ======

/-
α : Type u,
β : Type v,
f : α → β,
s : set α,
t : set β
⊢ f '' s ⊆ t ↔ s ⊆ f ⁻¹' t
  >> split,
| ⊢ f '' s ⊆ t → s ⊆ f ⁻¹' t
|   >> { intros h x xs,
| h : f '' s ⊆ t,
| x : α,
| xs : x ∈ s
| ⊢ x ∈ f ⁻¹' t
|   >>   show f x ∈ t,
|   >>     { calc f x ∈ f '' s : mem_image_of_mem f xs
|   >>            ... ⊆ t      : h }},
⊢ s ⊆ f ⁻¹' t → f '' s ⊆ t
  >> { intros h y hy,
h : s ⊆ f ⁻¹' t,
y : β,
hy : y ∈ f '' s
⊢ y ∈ t
  >>   rcases hy with ⟨x,xs,fxy⟩,
x : α,
xs : x ∈ s,
fxy : f x = y
⊢ y ∈ t
```

```
  >>   rw ← fxy,
⊢ f x ∈ t
  >>   apply h,
⊢ x ∈ s
  >>   exact xs },
no goals
-/
```

- Ejercicios de imágenes y uniones

```
import data.set.lattice

open set function

universes u1 u2 u3
variable  {α : Type u1}
variable  {β : Type u2}
variable  {I : Type u3}
variable  f : α → β
variable  A : I → set α
variable  B : I → set β

-- ---------------------------------------------------------------------
-- Ejercicio. Demostrar que
--    f '' (⋃ i, A i) = ⋃ i, f '' A i
-- ---------------------------------------------------------------------

example : f '' (⋃ i, A i) = ⋃ i, f '' A i :=
begin
  ext y,
  simp,
  split,
  { rintros ⟨x, ⟨i, xAi⟩, fxeq⟩,
    use [i, x, xAi, fxeq] },
  { rintros ⟨i, x, xAi, fxeq⟩,
    exact ⟨x, ⟨i, xAi⟩, fxeq⟩ },
end

-- Prueba
-- ======

/-
α : Type u1,
β : Type u2,
```

```
I : Type u3,
f : α → β,
A : I → set α
⊢ (f '' ⋃ (i : I), A i) = ⋃ (i : I), f '' A i
  >> ext y,
y : β
⊢ (y ∈ f '' ⋃ (i : I), A i) ↔ y ∈ ⋃ (i : I), f '' A i
  >> simp,
⊢ (∃ (x : α), (∃ (i : I), x ∈ A i) ∧ f x = y) ↔
  ∃ (i : I) (x : α), x ∈ A i ∧ f x = y
  >> split,
| ⊢ (∃ (x : α), (∃ (i : I), x ∈ A i) ∧ f x = y) →
|   (∃ (i : I) (x : α), x ∈ A i ∧ f x = y)
|   >> { rintros ⟨x, ⟨i, xAi⟩, fxeq⟩,
| x : α,
| fxeq : f x = y,
| i : I,
| xAi : x ∈ A i
| ⊢ ∃ (i : I) (x : α), x ∈ A i ∧ f x = y
|   >>   use [i, x, xAi, fxeq] },
⊢ (∃ (i : I) (x : α), x ∈ A i ∧ f x = y) →
  (∃ (x : α), (∃ (i : I), x ∈ A i) ∧ f x = y)
  >> { rintros ⟨i, x, xAi, fxeq⟩,
i : I,
x : α,
xAi : x ∈ A i,
fxeq : f x = y
⊢ ∃ (x : α), (∃ (i : I), x ∈ A i) ∧ f x = y
  >>   exact ⟨x, ⟨i, xAi⟩, fxeq⟩ },
no goals
-/

-- ---------------------------------------------------------------------
-- Ejercicio. Demostrar que
--    f '' (⋂ i, A i) ⊆ ⋂ i, f '' A i
-- ---------------------------------------------------------------------

example : f '' (⋂ i, A i) ⊆ ⋂ i, f '' A i :=
begin
  intro y,
  simp,
  intros x h fxeq i,
  use [x, h i, fxeq],
end
```

```
-- Prueba
-- ======

/-
α : Type u1,
β : Type u2,
I : Type u3,
f : α → β,
A : I → set α
⊢ (f '' ⋂ (i : I), A i) ⊆ ⋂ (i : I), f '' A i
  >> intro y,
y : β
⊢ (y ∈ f '' ⋂ (i : I), A i) → (y ∈ ⋂ (i : I), f '' A i)
  >> simp,
⊢ ∀ (x : α), (∀ (i : I), x ∈ A i) → f x = y →
  ∀ (i : I), ∃ (x : α), x ∈ A i ∧ f x = y
  >> intros x h fxeq i,
x : α,
h : ∀ (i : I), x ∈ A i,
fxeq : f x = y,
i : I
⊢ ∃ (x : α), x ∈ A i ∧ f x = y
  >> use [x, h i, fxeq],
no goals
-/

-- ---------------------------------------------------------------------
-- Ejercicio. Demostrar que si f es inyectiva e I no vacío, entonces
--    (⋂ i, f '' A i) ⊆ f '' (⋂ i, A i)
-- ---------------------------------------------------------------------

example
  (i : I)
  (injf : injective f)
  : (⋂ i, f '' A i) ⊆ f '' (⋂ i, A i) :=
begin
  intro y,
  simp,
  intro h,
  rcases h i with ⟨x, xAi, fxeq⟩,
  use x,
  split,
  { intro i',
    rcases h i' with ⟨x', x'Ai, fx'eq⟩,
    have : f x = f x', by rw [fxeq, fx'eq],
```

```
    have : x = x', from injf this,
    rw this,
    exact x'Ai },
  { exact fxeq },
end

-- Prueba
-- ======

/-
α : Type u1,
β : Type u2,
I : Type u3,
f : α → β,
A : I → set α,
i : I,
injf : injective f
⊢ (⋂ (i : I), f '' A i) ⊆ f '' ⋂ (i : I), A i
  >> intro y,
y : β
⊢ (y ∈ ⋂ (i : I), f '' A i) → (y ∈ f '' ⋂ (i : I), A i)
  >> simp,
⊢ (∀ (i : I), ∃ (x : α), x ∈ A i ∧ f x = y) →
  (∃ (x : α), (∀ (i : I), x ∈ A i) ∧ f x = y)
  >> intro h,
h : ∀ (i : I), ∃ (x : α), x ∈ A i ∧ f x = y
⊢ ∃ (x : α), (∀ (i : I), x ∈ A i) ∧ f x = y
  >> rcases h i with ⟨x, xAi, fxeq⟩,
x : α,
xAi : x ∈ A i,
fxeq : f x = y
⊢ ∃ (x : α), (∀ (i : I), x ∈ A i) ∧ f x = y
  >> use x,
⊢ (∀ (i : I), x ∈ A i) ∧ f x = y
  >> split,
| ⊢ ∀ (i : I), x ∈ A i
|   >> { intro i',
| i' : I
| ⊢ x ∈ A i'
|   >>   rcases h i' with ⟨x', x'Ai, fx'eq⟩,
| i' : I,
| x' : α,
| x'Ai : x' ∈ A i',
| fx'eq : f x' = y
| ⊢ x ∈ A i'
```

```
|   >>   have : f x = f x', by rw [fxeq, fx'eq],
| this : f x = f x'
| ⊢ x ∈ A i'
|   >>   have : x = x', from injf this,
| this : x = x'
| ⊢ x ∈ A i'
|   >>   rw this,
| ⊢ x' ∈ A i'
|   >>   exact x'Ai },
⊢ f x = y
  >> { exact fxeq },
no goals
-/

-- ---------------------------------------------------------------------
-- Ejercicio. Demostrar que
--    f ⁻¹' (⋃ i, B i) = ⋃ i, f ⁻¹' (B i)
-- ---------------------------------------------------------------------

example : f ⁻¹' (⋃ i, B i) = ⋃ i, f ⁻¹' (B i) :=
by { ext x, simp }

-- ---------------------------------------------------------------------
-- Ejercicio. Demostrar que
--    f ⁻¹' (⋂ i, B i) = ⋂ i, f ⁻¹' (B i)
-- ---------------------------------------------------------------------


example : f ⁻¹' (⋂ i, B i) = ⋂ i, f ⁻¹' (B i) :=
by { ext x, simp }
```

- Definición de inyectiva

```
import data.set.function

open set

universes u v
variable {α : Type u}
variable {β : Type v}
variable (f : α → β)
variable (s : set α)
```

```
-- ---------------------------------------------------------------------
-- Ejercicio. Demostrar que f es inyectiva sobre s syss
--    ∀ {x₁ x₂}, x₁ ∈ s → x₂ ∈ s → f x₁ = f x₂ → x₁ = x₂
-- ---------------------------------------------------------------------

example : inj_on f s ↔
  ∀ {x₁ x₂}, x₁ ∈ s → x₂ ∈ s → f x₁ = f x₂ → x₁ = x₂ :=
iff.refl _
```

- Inyectividad del logaritmo

```
-- ---------------------------------------------------------------------
-- Ejercicio. Demostrar que lo función logarítmica es inyectiva sobre
-- los números positivos.
-- ---------------------------------------------------------------------

import analysis.special_functions.exp_log

open set real

example : inj_on log { x | x > 0 } :=
begin
  intros x y xpos ypos,
  intro h,
  calc
    x   = exp (log x) : by rw exp_log xpos
    ... = exp (log y) : by rw h
    ... = y           : by rw exp_log ypos,
end

-- Prueba
-- ======

/-
⊢ inj_on log {x : ℝ | x > 0}
  >> intros x y xpos ypos,
x y : ℝ,
xpos : x ∈ {x : ℝ | x > 0},
ypos : y ∈ {x : ℝ | x > 0}
⊢ x.log = y.log → x = y
  >> intro h,
h : x.log = y.log
⊢ x = y
  >> calc
```

```
  >>   x   = exp (log x) : by rw exp_log xpos
  >>   ... = exp (log y) : by rw h
  >>   ... = y           : by rw exp_log ypos,
-/

-- Comentario: Se ha usado el lema
-- + exp_log : 0 < x → exp (log x) = x

-- Comprobación:
variable (x : ℝ)
#check @exp_log x
```

- Rango de la exponencial

```
-- ---------------------------------------------------------------------
-- Ejercicio. Demostrar que el rango de la función exponencial es el
-- conjunto de los números positivos,
-- ---------------------------------------------------------------------

import analysis.special_functions.exp_log

open set real

example : range exp = { y | y > 0 } :=
begin
  ext y,
  split,
  { rintros ⟨x, rfl⟩,
    apply exp_pos },
  { intro ypos,
    use log y,
    rw exp_log ypos },
end

-- Prueba
-- ======

/-
⊢ range exp = {y : ℝ | y > 0}
  >> ext y,
y : ℝ
⊢ y ∈ range exp ↔ y ∈ {y : ℝ | y > 0}
  >> split,
| y : ℝ
```

```
| ⊢ y ∈ range exp ↔ y ∈ {y : ℝ | y > 0}
|   >> { rintros ⟨x, rfl⟩
| x : ℝ
| ⊢ x.exp ∈ {y : ℝ | y > 0},
|   >>   apply exp_pos },
y : ℝ
⊢ y ∈ {y : ℝ | y > 0} → y ∈ range exp
  >> { intro ypos,
ypos : y ∈ {y : ℝ | y > 0}
⊢ y ∈ range exp
  >>   use log y,
⊢ y.log.exp = y
  >>   rw exp_log ypos },
⊢ y.log.exp = y
-/

-- Comentario: Se ha usado el lema
-- + exp_log : 0 < x → log (exp x) = x

variable (x : ℝ)
#check @exp_log x
```

- Inyectividad del cuadrado

```
import data.real.basic

open set real

-- ---------------------------------------------------------------------
-- Ejercicio. Demostrar que la función cuadrado es inyectiva sobre los
-- números no negativos.
-- ---------------------------------------------------------------------

example : inj_on sqrt { x | x ≥ 0 } :=
begin
  intros x y xnonneg ynonneg,
  intro e,
  calc
    x    = (sqrt x)^2 : by rw sqr_sqrt xnonneg
    ... = (sqrt y)^2 : by rw e
    ... = y          : by rw sqr_sqrt ynonneg,
end
```

```
-- Prueba
-- ======

/-
⊢ inj_on sqrt {x : ℝ | x ≥ 0}
  >> intros x y xnonneg ynonneg,
⊢ inj_on sqrt {x : ℝ | x ≥ 0}
  >> intro e,
e : x.sqrt = y.sqrt
⊢ x = y
  >> calc
  >>   x   = (sqrt x)^2 : by rw sqr_sqrt xnonneg
  >>   ... = (sqrt y)^2 : by rw e
  >>   ... = y          : by rw sqr_sqrt ynonneg,
no goals
-/

-- Comentario: Se ha usado el lema
-- + sqr_sqrt : 0 ≤ x → (sqrt x) ^ 2 = x

-- Comprobación:
variable (x : ℝ)
#check @sqr_sqrt x

example : inj_on (λ (x : ℝ), x^2) { x | x ≥ 0 } :=
begin
  intros x y xnonneg ynonneg,
  simp,
  intro e,
  calc
    x   = sqrt (x ^ 2) : by rw sqrt_sqr xnonneg
    ... = sqrt (y ^ 2) : by rw e
    ... = y            : by rw sqrt_sqr ynonneg,
end

-- Prueba
-- ======

/-
⊢ inj_on (λ (x : ℝ), x ^ 2) {x : ℝ | x ≥ 0}
  >> intros x y xnonneg ynonneg,
x y : ℝ,
xnonneg : x ∈ {x : ℝ | x ≥ 0},
ynonneg : y ∈ {x : ℝ | x ≥ 0}
⊢ (λ (x : ℝ), x ^ 2) x = (λ (x : ℝ), x ^ 2) y → x = y
```

```
  >> simp,
⊢ x ^ 2 = y ^ 2 → x = y
  >> intro e,
e : x ^ 2 = y ^ 2
⊢ x = y
  >> calc
  >>   x   = sqrt (x ^ 2) : by rw sqrt_sqr xnonneg
  >>   ... = sqrt (y ^ 2) : by rw e
  >>   ... = y            : by rw sqrt_sqr ynonneg,
no goals
-/

-- Comentario: Se ha usado el lema
-- + sqrt_sqr : 0 ≤ x → (x ^ 2).sqrt = x

#check @sqrt_sqr x
```

- Rango del cuadrado

```
import data.real.basic

open set real

-- ---------------------------------------------------------------------
-- Ejercicio. Demostrar que
--    sqrt '' { x | x ≥ 0 } = {y | y ≥ 0}
-- ---------------------------------------------------------------------

example : sqrt '' { x | x ≥ 0 } = {y | y ≥ 0} :=
begin
  ext,
  split,
  { intro h,
    rcases h with ⟨y,hy,eq⟩,
    simp at *,
    rw ← eq,
    exact sqrt_nonneg y },
  { intro h,
    use x ^ 2,
    simp at *,
    split,
    { exact pow_nonneg h 2 },
    { exact sqrt_sqr h }},
```

```
end

-- Prueba
-- ======

/-
⊢ sqrt '' {x : ℝ | x ≥ 0} = {y : ℝ | y ≥ 0}
  >> ext,
x : ℝ
⊢ x ∈ sqrt '' {x : ℝ | x ≥ 0} ↔ x ∈ {y : ℝ | y ≥ 0}
  >> split,
| x : ℝ
| ⊢ x ∈ sqrt '' {x : ℝ | x ≥ 0} → x ∈ {y : ℝ | y ≥ 0}
|   >> { intro h,
| h : x ∈ sqrt '' {x : ℝ | x ≥ 0}
| ⊢ x ∈ {y : ℝ | y ≥ 0}
|   >>   rcases h with ⟨y,hy,eq⟩,
| x y : ℝ,
| hy : y ∈ {x : ℝ | x ≥ 0},
| eq : y.sqrt = x
| ⊢ x ∈ {y : ℝ | y ≥ 0}
|   >>   simp at *,
| hy : 0 ≤ y
| ⊢ 0 ≤ x
|   >>   rw ← eq,
| ⊢ 0 ≤ y.sqrt
|   >>   exact sqrt_nonneg y },
x : ℝ
⊢ x ∈ {y : ℝ | y ≥ 0} → x ∈ sqrt '' {x : ℝ | x ≥ 0}
  >> { intro h,
h : x ∈ {y : ℝ | y ≥ 0}
⊢ x ∈ sqrt '' {x : ℝ | x ≥ 0}
  >>   use x ^ 2,
⊢ x ^ 2 ∈ {x : ℝ | x ≥ 0} ∧ (x ^ 2).sqrt = x
  >>   simp at *,
h : 0 ≤ x
⊢ 0 ≤ x ^ 2 ∧ (x ^ 2).sqrt = x
  >>   split,
| ⊢ 0 ≤ x ^ 2
|   >>   { exact pow_nonneg h 2 },
⊢ (x ^ 2).sqrt = x
  >>   { exact sqrt_sqr h }},
no goals
-/
```

```
-- Comentario: Se han usado los lemas
-- + x.sqrt_nonneg : 0 ≤ x.sqrt
-- + pow_nonneg : 0 ≤ x → ∀ (n : ℕ), 0 ≤ x ^ n

-- Comprobación:
variable (x : ℝ)
#check @sqrt_nonneg x
#check @pow_nonneg _ _ x

-- ---------------------------------------------------------------------
-- Ejercicio. Demostrar que
--    range (λ (x : ℝ), x^2) = {y | y ≥ 0} :=
-- ---------------------------------------------------------------------

example : range (λ (x : ℝ), x^2) = {y | y ≥ 0} :=
begin
  ext,
  split,
  { intro h,
    simp at *,
    rcases h with ⟨y,hy⟩,
    rw ← hy,
    exact pow_two_nonneg y },
  { intro h,
    use sqrt x,
    simp at *,
    exact sqr_sqrt h },
end

-- Prueba
-- ======

/-
⊢ range (λ (x : ℝ), x ^ 2) = {y : ℝ | y ≥ 0}
  >> ext,
x : ℝ
⊢ x ∈ range (λ (x : ℝ), x ^ 2) ↔ x ∈ {y : ℝ | y ≥ 0}
  >> split,
| ⊢ x ∈ range (λ (x : ℝ), x ^ 2) → x ∈ {y : ℝ | y ≥ 0}
|   >> { intro h,
| h : x ∈ range (λ (x : ℝ), x ^ 2)
| ⊢ x ∈ {y : ℝ | y ≥ 0}
|   >>   simp at *,
| h : ∃ (y : ℝ), y ^ 2 = x
| ⊢ 0 ≤ x
```

```
|   >>   rcases h with ⟨y,hy⟩,
| x y : ℝ,
| hy : y ^ 2 = x
| ⊢ 0 ≤ x
|   >>   rw ← hy,
| ⊢ 0 ≤ y ^ 2
|   >>   exact pow_two_nonneg y },
x : ℝ
⊢ x ∈ {y : ℝ | y ≥ 0} → x ∈ range (λ (x : ℝ), x ^ 2)
  >> { intro h,
h : x ∈ {y : ℝ | y ≥ 0}
⊢ x ∈ range (λ (x : ℝ), x ^ 2)
  >>   use sqrt x,
⊢ (λ (x : ℝ), x ^ 2) x.sqrt = x
  >>   simp at *,
h : 0 ≤ x
⊢ x.sqrt ^ 2 = x
  >>   exact sqr_sqrt h },
no goals
-/

-- Comentario: Se han usado los lemas
-- + pow_two_nonneg x : 0 ≤ x ^ 2
-- + sqr_sqrt : 0 ≤ x → (sqrt x) ^ 2 = x

-- Comprobación:
#check @pow_two_nonneg _ _ x
#check @sqr_sqrt x
```

- Ejercicios de imágenes y preimágenes

```
import data.set.function

open set function

universes u v
variable  α : Type u
variable  β : Type v
variable  f : α → β
variables s t : set α
variables u v : set β

-- ----------------------------------------------------------------------
```

```
-- Ejercicio. Demostrar que si f es inyectiva, entonces
--    f ⁻¹' (f '' s) ⊆ s
-- ----------------------------------------------------------------------

example
  (h : injective f)
  : f ⁻¹' (f '' s) ⊆ s :=
begin
  intros x hx,
  have h1 : f x ∈ f '' s := hx,
  rcases h1 with ⟨y, ys, fyfx⟩,
  have h2 : y = x := h fyfx,
  rw ← h2,
  exact ys,
end

-- Prueba
-- ======

/-
α : Type u,
β : Type v,
f : α → β,
s : set α,
h : injective f
⊢ f ⁻¹' (f '' s) ⊆ s
  >> intros x hx,
x : α,
hx : x ∈ f ⁻¹' (f '' s)
⊢ x ∈ s
  >> have h1 : f x ∈ f '' s := hx,
h1 : f x ∈ f '' s
⊢ x ∈ s
  >> rcases h1 with ⟨y, ys, fyfx⟩,
y : α,
ys : y ∈ s,
fyfx : f y = f x
⊢ x ∈ s
  >> have h2 : y = x := h fyfx,
h2 : y = x
⊢ x ∈ s
  >> rw ← h2,
⊢ y ∈ s
  >> exact ys,
no goals
```

```
-/

-- ---------------------------------------------------------------------
-- Ejercicio. Demostrar que
--    f '' (f⁻¹' u) ⊆ u
-- ---------------------------------------------------------------------

example : f '' (f⁻¹' u) ⊆ u :=
begin
  rintros y ⟨x, hxu, fxy⟩,
  rw ← fxy,
  exact hxu,
end

-- Prueba
-- ======

/-
α : Type u,
β : Type v,
f : α → β,
u : set β
⊢ f '' (f ⁻¹' u) ⊆ u
  >> rintros y ⟨x, hxu, fxy⟩,
y : β,
x : α,
hxu : x ∈ f ⁻¹' u,
fxy : f x = y
⊢ y ∈ u
  >> rw ← fxy,
⊢ f x ∈ u
  >> exact hxu,
no goals
-/

-- ---------------------------------------------------------------------
-- Ejercicio. Demostrar que si f es suprayectiva, entonces
--    u ⊆ f '' (f⁻¹' u)
-- ---------------------------------------------------------------------

example
  (h : surjective f)
  : u ⊆ f '' (f⁻¹' u) :=
begin
  intros y hy,
```

```
  rcases h y with ⟨x,hx⟩,
  use x,
  split,
  { simp,
    rw hx,
    exact hy},
  { exact hx },
end

-- Prueba
-- ======

/-
α : Type u,
β : Type v,
f : α → β,
u : set β,
h : surjective f
⊢ u ⊆ f '' (f ⁻¹' u)
  >> intros y hy,
y : β,
hy : y ∈ u
⊢ y ∈ f '' (f ⁻¹' u)
  >> rcases h y with ⟨x,hx⟩,
x : α,
hx : f x = y
⊢ y ∈ f '' (f ⁻¹' u)
  >> use x,
⊢ x ∈ f ⁻¹' u ∧ f x = y
  >> split,
| ⊢ x ∈ f ⁻¹' u
|   >> { simp,
| ⊢ f x ∈ u
|   >>   rw hx,
| ⊢ y ∈ u
|   >>   exact hy},
⊢ f x = y
  >> { exact hx },
-/

-- ---------------------------------------------------------------------
-- Ejercicio. Demostrar que si
--    s ⊆ t
-- entonces
--    f '' s ⊆ f '' t
```

```
-- ----------------------------------------------------------------------

example
  (h : s ⊆ t)
  : f '' s ⊆ f '' t :=
begin
  rintros y ⟨x,xs,fxy⟩,
  use x,
  split,
  { apply h,
    exact xs },
  { exact fxy },
end

-- Prueba
-- ======

/-
α : Type u,
β : Type v,
f : α → β,
s t : set α,
h : s ⊆ t
⊢ f '' s ⊆ f '' t
  >> rintros y ⟨x,xs,fxy⟩,
y : β,
x : α,
xs : x ∈ s,
fxy : f x = y
⊢ y ∈ f '' t
  >> use x,
⊢ x ∈ t ∧ f x = y
  >> split,
| ⊢ x ∈ t
|   >> { apply h,
| ⊢ x ∈ s
|   >>   exact xs },
⊢ f x = y
  >> { exact fxy },
no goals
-/

-- ----------------------------------------------------------------------
-- Ejercicio. Demostrar que si
--    u ⊆ v
```

```
-- entonces
--    f ⁻¹' u ⊆ f ⁻¹' v
-- ----------------------------------------------------------------------

example
  (h : u ⊆ v)
  : f ⁻¹' u ⊆ f ⁻¹' v :=
begin
  intros y hy,
  simp at *,
  apply h,
  exact hy,
end

-- Prueba
-- ======

/-
α : Type u,
β : Type v,
f : α → β,
u v : set β,
h : u ⊆ v
⊢ f ⁻¹' u ⊆ f ⁻¹' v
  >> intros y hy,
y : α,
hy : y ∈ f ⁻¹' u
⊢ y ∈ f ⁻¹' v
  >> simp at *,
hy : f y ∈ u
⊢ f y ∈ v
  >> apply h,
⊢ f y ∈ u
  >> exact hy,
no goals
-/

-- ----------------------------------------------------------------------
-- Ejercicio. Demostrar que
--    f ⁻¹' (u ∪ v) = (f ⁻¹' u) ∪ (f ⁻¹' v)
-- ----------------------------------------------------------------------

example : f ⁻¹' (u ∪ v) = (f ⁻¹' u) ∪ (f ⁻¹' v) :=
begin
  ext x,
```

```
  split,
  { intro hx,
    rcases hx with hxu | hxv,
    { left,
      apply hxu },
    { right,
      apply hxv }},
  { intro hx',
    simp,
    rcases hx' with hxu' | hxv',
    { left,
      apply hxu' },
    { right,
      apply hxv' }},
end

-- Prueba
-- ======

/-
α : Type u,
β : Type v,
f : α → β,
u v : set β
⊢ f ⁻¹' (u ∪ v) = f ⁻¹' u ∪ f ⁻¹' v
  >> ext x,
x : α
⊢ x ∈ f ⁻¹' (u ∪ v) ↔ x ∈ f ⁻¹' u ∪ f ⁻¹' v
  >> split,
| ⊢ x ∈ f ⁻¹' (u ∪ v) → x ∈ f ⁻¹' u ∪ f ⁻¹' v
|   >> { intro hx,
| hx : x ∈ f ⁻¹' (u ∪ v)
| ⊢ x ∈ f ⁻¹' u ∪ f ⁻¹' v
|   >>   rcases hx with hxu | hxv,
| | hxu : f x ∈ u
| | ⊢ x ∈ f ⁻¹' u ∪ f ⁻¹' v
| |   >>   { left,
| | ⊢ x ∈ f ⁻¹' u
| |   >>     apply hxu },
| ⊢ x ∈ f ⁻¹' u ∪ f ⁻¹' v
|   >>   { right,
| ⊢ x ∈ f ⁻¹' v
|   >>     apply hxv }},
⊢ x ∈ f ⁻¹' u ∪ f ⁻¹' v → x ∈ f ⁻¹' (u ∪ v)
  >> { intro hx',
```

```
hx' : x ∈ f ⁻¹' u ∪ f ⁻¹' v
⊢ x ∈ f ⁻¹' (u ∪ v)
  >>   simp,
⊢ f x ∈ u ∨ f x ∈ v
  >>   rcases hx' with hxu' | hxv',
| hxu' : x ∈ f ⁻¹' u
| ⊢ f x ∈ u ∨ f x ∈ v
|   >>   { left,
| ⊢ f x ∈ u
|   >>     apply hxu' },
hxv' : x ∈ f ⁻¹' v
⊢ f x ∈ u ∨ f x ∈ v
  >>   { right,
⊢ f x ∈ v
  >>     apply hxv' }},
no goals
-/

-- ---------------------------------------------------------------------
-- Ejercicio. Demostrar que
--    f '' (s ∩ t) ⊆ (f '' s) ∩ (f '' t)
-- ---------------------------------------------------------------------

example : f '' (s ∩ t) ⊆ (f '' s) ∩ (f '' t) :=
begin
  rintros y ⟨x,⟨xs,xt⟩,fxy⟩,
  split,
  { use x,
    exact ⟨xs, fxy⟩},
  { use x,
    exact ⟨xt, fxy⟩},
end

-- Prueba
-- ======

/-
α : Type u,
β : Type v,
f : α → β,
s t : set α
⊢ f '' (s ∩ t) ⊆ f '' s ∩ f '' t
  >> rintros y ⟨x,⟨xs,xt⟩,fxy⟩,
y : β,
x : α,
```

```
fxy : f x = y,
xs : x ∈ s,
xt : x ∈ t
⊢ y ∈ f '' s ∩ f '' t
  >> split,
| ⊢ y ∈ f '' s
|   >> { use x,
| ⊢ x ∈ s ∧ f x = y
|   >>   exact ⟨xs, fxy⟩},
⊢ y ∈ f '' t
  >> { use x,
⊢ x ∈ t ∧ f x = y
  >>   exact ⟨xt, fxy⟩},
no goals
-/

-- ---------------------------------------------------------------------
-- Ejercicio. Demostrar que si f es inyectiva, entonces
--    (f '' s) ∩ (f '' t) ⊆ f '' (s ∩ t)
-- ---------------------------------------------------------------------

example
  (h : injective f)
  : (f '' s) ∩ (f '' t) ⊆ f '' (s ∩ t) :=
begin
  rintros y ⟨ys,yt⟩,
  rcases ys with ⟨x,xs,fxy⟩,
  use x,
  split,
  { split,
    { exact xs },
    { rcases yt with ⟨z,zs,fzy⟩,
      rw ←fzy at fxy,
      rw h fxy,
      exact zs }},
  { exact fxy },
end

-- Prueba
-- ======

/-
α : Type u,
β : Type v,
f : α → β,
```

```
s t : set α,
h : injective f
⊢ f '' s ∩ f '' t ⊆ f '' (s ∩ t)
  >> rintros y ⟨ys,yt⟩,
y : β,
ys : y ∈ f '' s,
yt : y ∈ f '' t
⊢ y ∈ f '' (s ∩ t)
  >> rcases ys with ⟨x,xs,fxy⟩,
x : α,
xs : x ∈ s,
fxy : f x = y
⊢ y ∈ f '' (s ∩ t)
  >> use x,
⊢ x ∈ s ∩ t ∧ f x = y
  >> split,
| ⊢ x ∈ s ∩ t
|   >> { split,
| | ⊢ x ∈ s
| |   >>   { exact xs },
| ⊢ x ∈ t
|   >>   { rcases yt with ⟨z,zs,fzy⟩,
| z : α,
| zs : z ∈ t,
| fzy : f z = y
| ⊢ x ∈ t
|   >>     rw ←fzy at fxy,
| fxy : f x = f z
| ⊢ x ∈ t
|   >>     rw h fxy,
| fxy : f x = f z
| ⊢ z ∈ t
|   >>     exact zs }},
⊢ f x = y
  >> { exact fxy },
no goals
-/

-- ---------------------------------------------------------------------
-- Ejercicio. Demostrar que
--    (f '' s) \ (f '' t) ⊆ f '' (s \ t)
-- ---------------------------------------------------------------------

example : (f '' s) \ (f '' t) ⊆ f '' (s \ t) :=
begin
```

```
  rintros y ⟨h1,h2⟩,
  rcases h1 with ⟨x,xs,fxy⟩,
  use x,
  split,
  { split,
    { exact xs },
    { intro h3,
      apply h2,
      use x,
      exact ⟨h3, fxy⟩}},
  { exact fxy },
end

-- Prueba
-- ======

/-
α : Type u,
β : Type v,
f : α → β,
s t : set α
⊢ f '' s \ f '' t ⊆ f '' (s \ t)
  >> rintros y ⟨h1,h2⟩,
y : β,
h1 : y ∈ f '' s,
h2 : y ∉ f '' t
⊢ y ∈ f '' (s \ t)
  >> rcases h1 with ⟨x,xs,fxy⟩,
x : α,
xs : x ∈ s,
fxy : f x = y
⊢ y ∈ f '' (s \ t)
  >> use x,
⊢ x ∈ s \ t ∧ f x = y
  >> split,
| ⊢ x ∈ s \ t
|   >> { split,
| ⊢ x ∈ s
| |   >>   { exact xs },
| ⊢ (λ (a : α), a ∉ t) x
|   >>   { intro h3,
| h3 : x ∈ t
| ⊢ false
|   >>     apply h2,
| ⊢ y ∈ f '' t
```

```
|   >>     use x,
| ⊢ x ∈ t ∧ f x = y
|   >>     exact ⟨h3, fxy⟩}},
⊢ f x = y
  >> { exact fxy },
no goals
-/

-- ---------------------------------------------------------------------
-- Ejercicio. Demostrar que
--    (f ⁻¹' u) \ (f ⁻¹' v) ⊆ f ⁻¹' (u \ v)
-- ---------------------------------------------------------------------

example : (f ⁻¹' u) \ (f ⁻¹' v) ⊆ f ⁻¹' (u \ v) :=
begin
  rintros x ⟨hxu,hxv⟩,
  simp,
  split,
  { exact hxu },
  { intro h,
    apply hxv,
    exact h },
end

-- Prueba
-- ======

/-
α : Type u,
β : Type v,
f : α → β,
u v : set β
⊢ f ⁻¹' u \ f ⁻¹' v ⊆ f ⁻¹' (u \ v)
  >> rintros x ⟨hxu,hxv⟩,
x : α,
hxu : x ∈ f ⁻¹' u,
hxv : x ∉ f ⁻¹' v
⊢ x ∈ f ⁻¹' (u \ v)
  >> simp,
⊢ f x ∈ u ∧ f x ∉ v
  >> split,
| ⊢ f x ∈ u
|   >> { exact hxu },
⊢ f x ∉ v
  >> { intro h,
```

```
h : f x ∈ v
⊢ false
  >>   apply hxv,
⊢ x ∈ f ⁻¹' v
  >>   exact h },
no goals
-/

-- ---------------------------------------------------------------------
-- Ejercicio. Demostrar que
--    (f '' s) ∩ v = f '' (s ∩ (f ⁻¹' v))
-- ---------------------------------------------------------------------

example : (f '' s) ∩ v = f '' (s ∩ (f ⁻¹' v)) :=
begin
  ext y,
  split,
  { rintros ⟨ys,yv⟩,
    rcases ys with ⟨x,xs,fxy⟩,
    use x,
    split,
    { split,
      { exact xs },
      { simp,
        rw fxy,
        exact yv }},
    { exact fxy }},
  { intro h,
    rcases h with ⟨x,⟨hxs,hxv⟩,fxy⟩,
    split,
    { rw ←fxy,
      exact mem_image_of_mem f hxs },
    { rw ←fxy,
      exact hxv }},
end

-- Prueba
-- ======

/-
α : Type u,
β : Type v,
f : α → β,
s : set α,
v : set β
```

```
⊢ f '' s ∩ v = f '' (s ∩ f ⁻¹' v)
  >> ext y,
y : β
⊢ y ∈ f '' s ∩ v ↔ y ∈ f '' (s ∩ f ⁻¹' v)
  >> split,
| ⊢ y ∈ f '' s ∩ v → y ∈ f '' (s ∩ f ⁻¹' v)
|   >> { rintros ⟨ys,yv⟩,
| ys : y ∈ f '' s,
| yv : y ∈ v
| ⊢ y ∈ f '' (s ∩ f ⁻¹' v)
|   >>   rcases ys with ⟨x,xs,fxy⟩,
| x : α,
| xs : x ∈ s,
| fxy : f x = y
| ⊢ y ∈ f '' (s ∩ f ⁻¹' v)
|   >>   use x,
⊢ x ∈ s ∩ f ⁻¹' v ∧ f x = y
|   >>   split,
| ⊢ x ∈ s ∩ f ⁻¹' v
|   >>   { split,
| |  ⊢ x ∈ s
| |    >>     { exact xs },
| ⊢ x ∈ f ⁻¹' v
|   >>     { simp,
| ⊢ f x ∈ v
|   >>       rw fxy,
| ⊢ y ∈ v
|   >>       exact yv }},
| ⊢ f x = y
|   >>   { exact fxy }},
⊢ y ∈ f '' (s ∩ f ⁻¹' v) → y ∈ f '' s ∩ v
  >> { intro h,
h : y ∈ f '' (s ∩ f ⁻¹' v)
⊢ y ∈ f '' s ∩ v
  >>   rcases h with ⟨x,⟨hxs,hxv⟩,fxy⟩,
x : α,
fxy : f x = y,
hxs : x ∈ s,
hxv : x ∈ f ⁻¹' v
⊢ y ∈ f '' s ∩ v
  >>   split,
| ⊢ y ∈ f '' s
|   >>   { rw ←fxy,
| ⊢ f x ∈ f '' s
|   >>     exact mem_image_of_mem f hxs },
```

```
⊢ y ∈ v
  >>   { rw ←fxy,
⊢ f x ∈ v
  >>     exact hxv }},
no goals
-/

-- ---------------------------------------------------------------------
-- Ejercicio. Demostrar que
--    f '' (s ∩ f ⁻¹' u) ⊆ (f '' s) ∪ u
-- ---------------------------------------------------------------------

example : f '' (s ∩ f ⁻¹' u) ⊆ (f '' s) ∪ u :=
begin
  intros y h,
  rcases h with ⟨x,⟨xs,xu⟩,fxy⟩,
  right,
  rw ←fxy,
  exact xu,
end

-- Prueba
-- ======

/-
α : Type u,
β : Type v,
f : α → β,
s : set α,
u : set β
⊢ f '' (s ∩ f ⁻¹' u) ⊆ f '' s ∪ u
  >> intros y h,
y : β,
h : y ∈ f '' (s ∩ f ⁻¹' u)
⊢ y ∈ f '' s ∪ u
  >> rcases h with ⟨x,⟨xs,xu⟩,fxy⟩,
x : α,
fxy : f x = y,
xs : x ∈ s,
xu : x ∈ f ⁻¹' u
⊢ y ∈ f '' s ∪ u
  >> right,
⊢ y ∈ u
  >> rw ←fxy,
⊢ f x ∈ u
```

```
  >> exact xu,
no goals
-/

-- ---------------------------------------------------------------------
-- Ejercicio. Demostrar que
--    s ∩ f ⁻¹' u ⊆ f ⁻¹' ((f '' s) ∩ u) :=
-- ---------------------------------------------------------------------

example : s ∩ f ⁻¹' u ⊆ f ⁻¹' ((f '' s) ∩ u) :=
begin
  rintros x ⟨xs,xu⟩,
  simp at xu,
  split,
  { exact mem_image_of_mem f xs },
  { exact xu },
end

-- Prueba
-- ======

/-
α : Type u,
β : Type v,
f : α → β,
s : set α,
u : set β
⊢ s ∩ f ⁻¹' u ⊆ f ⁻¹' (f '' s ∩ u)
  >> rintros x ⟨xs,xu⟩,
x : α,
xs : x ∈ s,
xu : x ∈ f ⁻¹' u
⊢ x ∈ f ⁻¹' (f '' s ∩ u)
  >> simp at xu,
xu : f x ∈ u
⊢ x ∈ f ⁻¹' (f '' s ∩ u)
  >> split,
| ⊢ f x ∈ f '' s
|   >> { exact mem_image_of_mem f xs },
⊢ f x ∈ u
  >> { exact xu },
no goals
-/

-- ---------------------------------------------------------------------
```

```
-- Ejercicio. Demostrar que
--    s ∪ f ⁻¹' u ⊆ f ⁻¹' ((f '' s) ∪ u)
-- ----------------------------------------------------------------------

example : s ∪ f ⁻¹' u ⊆ f ⁻¹' ((f '' s) ∪ u) :=
begin
  rintros x (xs | xu),
  { left,
    exact mem_image_of_mem f xs },
  { right,
    exact xu },
end

-- Prueba
-- ======

/-
α : Type u,
β : Type v,
f : α → β,
s : set α,
u : set β
⊢ s ∪ f ⁻¹' u ⊆ f ⁻¹' (f '' s ∪ u)
  >> rintros x (xs | xu),
| x : α,
| xs : x ∈ s
| ⊢ x ∈ f ⁻¹' (f '' s ∪ u)
|   >> { left,
| ⊢ f x ∈ f '' s
|   >>   exact mem_image_of_mem f xs },
xu : x ∈ f ⁻¹' u
⊢ x ∈ f ⁻¹' (f '' s ∪ u)
  >> { right,
⊢ f x ∈ u
  >>   exact xu },
no goals
-/
```

- Valor por defecto y elección de valores

```
-- ----------------------------------------------------------------------
-- Ejercicio. Declarar α como una variables de tipos habitados.
-- ----------------------------------------------------------------------
```

```
variables {α : Type*} [inhabited α]

-- ---------------------------------------------------------------------
-- Ejercicio. Calcular el tipo de
--     default α
-- ---------------------------------------------------------------------

#check default α

-- Comntario: Al colocar el cursor sobre check se obtiene
--    default α : α

-- ---------------------------------------------------------------------
-- Ejercicio. Declarar P como un predicado sobre α tal que existe algún
-- elemento que verifica P.
-- ---------------------------------------------------------------------

variables (P : α → Prop) (h : ∃ x, P x)

-- ---------------------------------------------------------------------
-- Ejercicio. Calcular el tipo de
--    classical.some h
-- ---------------------------------------------------------------------

#check classical.some h

-- Comntario: Al colocar el cursor sobre check se obtiene
--    classical.some h : α

-- ---------------------------------------------------------------------
-- Ejercicio. Demostrar que
--    P (classical.some h)
-- ---------------------------------------------------------------------

example : P (classical.some h) :=
classical.some_spec h
```

- Función inversa

```
-- ---------------------------------------------------------------------
-- Ejercicio. Realizar las siguientes acciones:
-- 1. Importar la teoría data.set.function
-- 2. Declarar u y v como universos.
-- 3. Declarar α como variable sobre tipo de u habitados.
```

```
-- 4. Declarar β como variable sobre tipo de v.
-- 5. Declarar la teoría como no computable.
-- 6. Usar la lógica clásica.
-- ---------------------------------------------------------------------

import data.set.function               -- 1
universes u v                          -- 2
variables {α : Type u} [inhabited α]   -- 3
variables {β : Type v}                 -- 4
noncomputable theory                   -- 5
open_locale classical                  -- 6

-- ---------------------------------------------------------------------
-- Ejercicio. Definir la inversa de una función
-- ---------------------------------------------------------------------

def inverse (f : α → β) : β → α :=
λ y : β, if h : ∃ x, f x = y
         then classical.some h
         else default α

-- ---------------------------------------------------------------------
-- Ejercicio. Sea d una función de α en β e y un elemento de
-- β. Demostrar que si
--    ∃ x, f x = y
-- entonces
--    f (inverse f y) = y :=
-- ---------------------------------------------------------------------

theorem inverse_spec
  {f : α → β}
  (y : β)
  (h : ∃ x, f x = y) :
  f (inverse f y) = y :=
begin
  rw inverse, dsimp, rw dif_pos h,
  exact classical.some_spec h
end

-- Prueba
-- ======

/-
α : Type u,
_inst_1 : inhabited α,
```

```
β : Type v,
f : α → β,
y : β,
h : ∃ (x : α), f x = y
⊢ f (inverse f y) = y
  >> rw inverse, dsimp, rw dif_pos h,
⊢ f (classical.some h) = y
  >> exact classical.some_spec h,
no goals
-/

-- Comentarios:
-- 1. La identidad (dif_pos h), cuando (h : e), reescribe la expresión
--    (if h : e then x else y) a x.
-- 2. La identidad (dif_neg h), cuando (h : ¬ e), reescribe la expresión
--    (if h : e then x else y) a y.
```

- Caracterización de las funciones inyectivas mediante la inversa por la izquierda

```
import .Funcion_inversa

universes u v
variables {α : Type u} [inhabited α]
variables {β : Type v}
variable  f : α → β
variable  g : β → α
variable  x : α

open set function

-- ---------------------------------------------------------------------
-- Ejercicio. Demostrar que g es la inversa por la izquierda de f syss
--    ∀ x, g (f x) = x
-- ---------------------------------------------------------------------

example : left_inverse g f ↔ ∀ x, g (f x) = x :=
by rw left_inverse

-- ---------------------------------------------------------------------
-- Ejercicio. Demostrar que las siguientes condiciones son equivalentes:
-- 1. f es inyectiva
-- 2. left_inverse (inverse f) f
```

```
-- ----------------------------------------------------------------------

-- 1ª demostración
-- ===============

example : injective f ↔ left_inverse (inverse f) f  :=
begin
  split,
  { intros h y,
    apply h,
    apply inverse_spec,
    use y },
  { intros h x1 x2 e,
    rw ←h x1,
    rw ←h x2,
    rw e },
end

-- Prueba
-- ======

/-
α : Type u,
_inst_1 : inhabited α,
β : Type v,
f : α → β
⊢ injective f ↔ left_inverse (inverse f) f
  >> split,
| ⊢ injective f → left_inverse (inverse f) f
|   >> { intros h y,
| h : injective f,
| y : α
| ⊢ inverse f (f y) = y
|   >>   apply h,
| ⊢ f (inverse f (f y)) = f y
|   >>   apply inverse_spec,
| ⊢ ∃ (x : α), f x = f y
|   >>   use y },
⊢ left_inverse (inverse f) f → injective f
  >> { intros h x1 x2 e,
h : left_inverse (inverse f) f,
x1 x2 : α,
e : f x1 = f x2
⊢ x1 = x2
  >>   rw ←h x1,
```

```
⊢ inverse f (f x1) = x2
  >>   rw ←h x2,
⊢ inverse f (f x1) = inverse f (f x2)
  >>   rw e },
no goals
-/

-- 2ª demostración
-- ===============

example : injective f ↔ left_inverse (inverse f) f  :=
⟨λ h y, h (inverse_spec _ ⟨y, rfl⟩),
 λ h x1 x2 e, by rw [←h x1, ←h x2, e]⟩
```

- Caracterización de las funciones suprayectivas mediante la inversa por la derecha

```
import .Funcion_inversa

universes u v
variables {α : Type u} [inhabited α]
variables {β : Type v}
variable  f : α → β
variable  g : β → α
variable  x : α

open set function

-- ---------------------------------------------------------------------
-- Ejercicio. Demostrar que g es la inversa por la derecha de f syss
--    ∀ x, f (g x) = x :=
-- ---------------------------------------------------------------------

example : right_inverse g f ↔ ∀ x, f (g x) = x :=
begin
  rw function.right_inverse,
  rw left_inverse,
end

-- ---------------------------------------------------------------------
-- Ejercicio. Demostrar que las siguientes condiciones son equivalentes:
-- 1. f es suprayectiva
-- 2. right_inverse (inverse f) f
```

```
-- ---------------------------------------------------------------------

-- 1ª demostración
-- ===============

example : surjective f ↔ right_inverse (inverse f) f :=
begin
  split,
  { intros h y,
    apply inverse_spec,
    apply h },
  { intros h y,
    use (inverse f y),
    apply h },
end

-- Prueba
-- ======

/-
α : Type u,
_inst_1 : inhabited α,
β : Type v,
f : α → β
⊢ surjective f ↔ right_inverse (inverse f) f
  >> split,
| ⊢ surjective f → right_inverse (inverse f) f
|   >> { intros h y,
| h : surjective f,
| y : β
| ⊢ f (inverse f y) = y
|   >>   apply inverse_spec,
| ⊢ ∃ (x : α), f x = y
|   >>   apply h },
⊢ right_inverse (inverse f) f → surjective f
  >> { intros h y,
h : right_inverse (inverse f) f,
y : β
⊢ ∃ (a : α), f a = y
  >>   use (inverse f y),
⊢ f (inverse f y) = y
  >>   apply h },
no goals
-/
```

```
-- 2ª demostración
-- ===============

example : surjective f ↔ right_inverse (inverse f) f :=
⟨λ h y, inverse_spec _ (h _),
 λ h y, ⟨inverse f y, h _⟩⟩

-- ---------------------------------------------------------------------
-- Ejercicio. Demostrar que f es suprayectiva syss tiene una inversa por
-- la izquierda.
-- ---------------------------------------------------------------------

example : surjective f ↔ ∃ g, right_inverse g f :=
begin
  split,
  { intro h,
    dsimp [surjective] at h,
    choose g hg using h,
    use g,
    exact hg },
  { rintro ⟨g, hg⟩,
    intros b,
    use g b,
    exact hg b },
end

-- Prueba
-- ======

/-
α : Type u,
_inst_1 : inhabited α,
β : Type v,
f : α → β
⊢ surjective f ↔ ∃ (g : β → α), right_inverse g f
  >> split,
| ⊢ surjective f → (∃ (g : β → α), right_inverse g f)
|   >> { intro h,
| h : surjective f
| ⊢ ∃ (g : β → α), right_inverse g f
|   >>   dsimp [surjective] at h,
| h : ∀ (b : β), ∃ (a : α), f a = b
| ⊢ ∃ (g : β → α), right_inverse g f
|   >>   choose g hg using h,
| g : β → α,
```

```
| hg : ∀ (b : β), f (g b) = b
| ⊢ ∃ (g : β → α), right_inverse g f
|   >>   use g,
| ⊢ right_inverse g f
|   >>   exact hg },
⊢ (∃ (g : β → α), right_inverse g f) → surjective f
  >> { rintro ⟨g, hg⟩,
g : β → α,
hg : right_inverse g f
⊢ surjective f
  >>   intros b,
b : β
⊢ ∃ (a : α), f a = b
  >>   use g b,
⊢ f (g b) = b
  >>   exact hg b },
no goals
-/

-- Comentantarios:
-- 1. La táctica (dsimp [e] at h) simplifica la hipótesis h con la
--    definición de e.
-- 2. La táctica (choose g hg using h) si h es de la forma
--    (∀ (b : β), ∃ (a : α), f a = b) quita la hipótesis h y añade las
--     hipótesis (g : β → α) y (hg : ∀ (b : β), f (g b) = b).
```

▪ Teorema de Cantor

```
-- ----------------------------------------------------------------------
-- Ejercicio. Demostrar el teorema de Cantor: No existe singuna
-- aplicación suprayectiva de un conjunto en su conjunto potencia.
-- ----------------------------------------------------------------------

import data.set.basic

open function

variable {α : Type*}

theorem Cantor : ∀ f : α → set α, ¬ surjective f :=
begin
  intros f surjf,
  let S := { i | i ∉ f i},
  rcases surjf S with j,
```

```
  have h₁ : j ∉ f j,
  { intro h',
    have : j ∉ f j,
      { by rwa h at h' },
    contradiction },
  have h₂ : j ∈ S,
    from h₁,
  have h₃ : j ∉ S,
    by rwa h at h₁,
  contradiction,
end

-- Prueba
-- ======

/-
α : Type u_1
⊢ ∀ (f : α → set α), ¬surjective f
  >> intros f surjf,
f : α → set α,
surjf : surjective f
⊢ false
  >> let S := { i | i ∉ f i},
S : set α := {i : α | i ∉ f i}
⊢ false
  >> rcases surjf S with j,
j : α,
h : f j = S
⊢ false
  >> have h₁ : j ∉ f j,
| ⊢ j ∉ f j
|   >> { intro h',
| h' : j ∈ f j
| ⊢ false
|   >>   have : j ∉ f j,
| | ⊢ j ∉ f j
| |   >>     { by rwa h at h' },
| this : j ∉ f j
| ⊢ false
|   >>   contradiction },
h₁ : j ∉ f j
⊢ false
  >> have h₂ : j ∈ S,
| ⊢ j ∈ S
|   >>   from h₁,
```

```
h₂ : j ∈ S
⊢ false
  >> have h₃ : j ∉ S,
  >>   by rwa h at h₁,
h₃ : j ∉ S
⊢ false
  >> contradiction,
no goals
-/
```

Capítulo 5

Bibliografía

- Abstract algebra cheatsheet. de Matthias Vallentin.
- Basic guide to tactics de Kevin Buzzard.
- Lean mathlib documentation.
- Mathematics in Lean de Jeremy Avigad, Kevin Buzzard, Robert Y. Lewis y Patrick Massot.
- Maths challenges for the Lean curious de Kevin Buzzard.
- The Lean reference manual de Jeremy Avigad, Gabriel Ebner y Sebastian Ullrich.