Calculemus
(Ejercicios de demostración con Lean)

José A. Alonso Jiménez

Grupo de Lógica Computacional
Dpto. de Ciencias de la Computación e Inteligencia Artificial
Universidad de Sevilla
Sevilla, 7 de agosto de 2021

Índice general

Capítulo 1

Introducción

En el blog Calculemus se han ido proponiendo ejercicios de demostración de resultados matemáticos usando sistemas de demostración interactiva.

En este libro se hace una recopilación de dichos ejercicios usando Lean (concretamente, con su versión 3.30.0). La ordenación de los ejercicios es simplemente temporal según su fecha de publicación en Calculemus. En futuras versiones del libro está previsto cambiar la ordenación por otra temática.

Por otra parte, este libro es una continuación del DAO (Demostración Asistida por Ordenador) con Lean con el que comparte el objetivo de usarse en las clases de la asignatura de Razonamiento automático del Máster Universitario en Lógica, Computación e Inteligencia Artificial de la Universidad de Sevilla. Por tanto, el único prerrequisito es, como en el Máster, cierta madurez matemática como la que deben tener los alumnos de los Grados de Matemática y de Informática.

En cada ejercicio, se exponen distintas soluciones ordenadas desde las más detalladas a las más automáticas y se proporciona un enlace que añ pulsarlo abre el ejercicio en Lean Web (en una sesión del navegador) de forma que se puede navegar por las pruebas y editar otras alternativas

Las soluciones del libro están en este repositorio de GitHub.

Capítulo 2

Ejercicios de mayo de 2021

2.1. Propiedad de monotonía de la intersección

```
-- ----------------------------------------------------------------------
-- Demostrar que si
--    s ⊆ t
-- entonces
--    s ∩ u ⊆ t ∩ u
-- ----------------------------------------------------------------------

import data.set.basic
open set

variable {α : Type}
variables s t u : set α

-- 1ª demostración
-- ===============

example
  (h : s ⊆ t)
  : s ∩ u ⊆ t ∩ u :=
begin
  rw subset_def,
  rw inter_def,
  rw inter_def,
  dsimp,
  intros x h,
  cases h with xs xu,
  split,
  { rw subset_def at h,
```

```
    apply h,
    assumption },
  { assumption },
end

-- 2ª demostración
-- ===============

example
  (h : s ⊆ t)
  : s ∩ u ⊆ t ∩ u :=
begin
  rw [subset_def, inter_def, inter_def],
  dsimp,
  rintros x ⟨xs, xu⟩,
  rw subset_def at h,
  exact ⟨h _ xs, xu⟩,
end

-- 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

-- 4ª demostración
-- ===============

example
  (h : s ⊆ t)
  : s ∩ u ⊆ t ∩ u :=
begin
  intros x xsu,
  exact ⟨h xsu.1, xsu.2⟩,
end

-- 5ª demostración
-- ===============
```

```
example
  (h : s ⊆ t)
  : s ∩ u ⊆ t ∩ u :=
inter_subset_inter_left u h
```

Se puede interactuar con las pruebas anteriores en esta sesión con Lean.

2.2. Propiedad semidistributiva de la intersección sobre la unión

```
-- ---------------------------------------------------------------------
-- Demostrar que
--    s ∩ (t ∪ u) ⊆ (s ∩ t) ∪ (s ∩ u)
-- ----------------------------------------------------------------------

import data.set.basic
open set

variable {α : Type}
variables s t u : set α

-- 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,
  clear hx,
  cases xtu with xt xu,
  { left,
    show x ∈ s ∩ t,
    exact ⟨xs, xt⟩ },
  { right,
    show x ∈ s ∩ u,
    exact ⟨xs, xu⟩ },
end

-- 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

-- 3ª demostración
-- ===============

example :
  s ∩ (t ∪ u) ⊆ (s ∩ t) ∪ (s ∩ u) :=
begin
  intros x hx,
  by finish
end

-- 4ª demostración
-- ===============

example :
  s ∩ (t ∪ u) ⊆ (s ∩ t) ∪ (s ∩ u) :=
by rw inter_union_distrib_left
```

Se puede interactuar con las pruebas anteriores en esta sesión con Lean.

2.3. Diferencia de diferencia de conjuntos

```
-- ------------------------------------------------------------------------
-- Demostrar que
--    (s \ t) \ u ⊆ s \ (t ∪ u)
-- -------------------------------------------------------------------------

import data.set.basic
open set

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

-- 2ª demostración
-- ===============

example : (s \ t) \ u ⊆ s \ (t ∪ u) :=
begin
  rintros x ⟨⟨xs, xnt⟩, xnu⟩,
  use xs,
  rintros (xt | xu); contradiction
end

-- 3ª demostración
-- ===============

example : (s \ t) \ u ⊆ s \ (t ∪ u) :=
begin
  intros x xstu,
  simp at *,
  finish,
end

-- 4ª demostración
-- ===============

example : (s \ t) \ u ⊆ s \ (t ∪ u) :=
begin
  intros x xstu,
```

```
  finish,
end

-- 5ª demostración
-- ===============

example : (s \ t) \ u ⊆ s \ (t ∪ u) :=
by rw diff_diff
```

Se puede interactuar con las pruebas anteriores en esta sesión con Lean.

2.4. 2ª propiedad semidistributiva de la intersección sobre la unión

```
-- ----------------------------------------------------------------------
-- Demostrar que
--    (s ∩ t) ∪ (s ∩ u) ⊆ s ∩ (t ∪ u)
-- -----------------------------------------------------------------------

import data.set.basic
open set

variable {α : Type}
variables s t u : set α

-- 1ª demostración
-- ===============

example : (s ∩ t) ∪ (s ∩ u) ⊆ s ∩ (t ∪ u):=
begin
  intros x hx,
  cases hx with xst xsu,
  { split,
    { exact xst.1 },
    { left,
      exact xst.2 }},
  { split,
    { exact xsu.1 },
    { right,
      exact xsu.2 }},
end
```

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

example : (s ∩ t) ∪ (s ∩ u) ⊆ s ∩ (t ∪ u):=
begin
  rintros x (⟨xs, xt⟩ | ⟨xs, xu⟩),
  { use xs,
    left,
    exact xt },
  { use xs,
    right,
    exact xu },
end

-- 3ª demostración
-- ===============

example : (s ∩ t) ∪ (s ∩ u) ⊆ s ∩ (t ∪ u):=
by rw inter_distrib_left s t u

-- 4ª demostración
-- ===============

example : (s ∩ t) ∪ (s ∩ u) ⊆ s ∩ (t ∪ u):=
begin
  intros x hx,
  finish
end
```

Se puede interactuar con las pruebas anteriores en esta sesión con Lean.

2.5. 2ª diferencia de diferencia de conjuntos

```
-- ---------------------------------------------------------------------
-- Demostrar que
--    s \ (t ∪ u) ⊆ (s \ t) \ u
-- ----------------------------------------------------------------------

import data.set.basic
open set

variable {α : Type}
variables s t u : set α
```

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

example : s \ (t ∪ u) ⊆ (s \ t) \ u :=
begin
  intros x hx,
  split,
  { split,
    { exact hx.1, },
    { dsimp,
      intro xt,
      apply hx.2,
      left,
      exact xt, }},
  { dsimp,
    intro xu,
    apply hx.2,
    right,
    exact xu, },
end

-- 2ª demostración
-- ===============

example : s \ (t ∪ u) ⊆ (s \ t) \ u :=
begin
  rintros x ⟨xs, xntu⟩,
  split,
  { split,
    { exact xs, },
    { intro xt,
      exact xntu (or.inl xt), }},
  { intro xu,
    exact xntu (or.inr xu), },
end

-- 3ª demostración
-- ===============

example : s \ (t ∪ u) ⊆ (s \ t) \ u :=
begin
  rintros x ⟨xs, xntu⟩,
  use xs,
  { intro xt,
```

```
    exact xntu (or.inl xt) },
  { intro xu,
    exact xntu (or.inr xu) },
end

-- 4ª demostración
-- ===============

example : s \ (t ∪ u) ⊆ (s \ t) \ u :=
begin
  rintros x ⟨xs, xntu⟩;
  finish,
end

-- 5ª demostración
-- ===============

example : s \ (t ∪ u) ⊆ (s \ t) \ u :=
by intro ; finish

-- 6ª demostración
-- ===============

example : s \ (t ∪ u) ⊆ (s \ t) \ u :=
by rw diff_diff
```

Se puede interactuar con las pruebas anteriores en esta sesión con Lean.

2.6. Conmutatividad de la intersección

```
-- ----------------------------------------------------------------------
-- Demostrar que
--    s ∩ t = t ∩ s
-- ----------------------------------------------------------------------

import data.set.basic
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,
  { intro h,
    split,
    { exact h.2, },
    { exact h.1, }},
  { intro h,
    split,
    { exact h.2, },
    { exact h.1, }},
end

-- 2ª demostración
-- ===============

example : s ∩ t = t ∩ s :=
begin
  ext,
  simp only [mem_inter_eq],
  exact ⟨λ h, ⟨h.2, h.1⟩,
         λ h, ⟨h.2, h.1⟩⟩,
end

-- 3ª demostración
-- ===============

example : s ∩ t = t ∩ s :=
begin
  ext,
  exact ⟨λ h, ⟨h.2, h.1⟩,
         λ h, ⟨h.2, h.1⟩⟩,
end

-- 4ª 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

-- 5ª demostración
-- ===============

example : s ∩ t = t ∩ s :=
begin
  ext x,
  exact and.comm,
end

-- 6ª demostración
-- ===============

example : s ∩ t = t ∩ s :=
ext (λ x, and.comm)

-- 7ª demostración
-- ===============

example : s ∩ t = t ∩ s :=
by ext x; simp [and.comm]

-- 8ª demostración
-- ===============

example : s ∩ t = t ∩ s :=
inter_comm s t

-- 9ª demostración
-- ===============

example : s ∩ t = t ∩ s :=
by finish
```

Se puede interactuar con las pruebas anteriores en esta sesión con Lean.

2.7. Intersección con su unión

```
-- ----------------------------------------------------------------------
-- Demostrar que
--    s ∩ (s ∪ t) = s
-- ----------------------------------------------------------------------

import data.set.basic
open set

variable {α : Type}
variables s t : set α

-- 1ª demostración
-- ===============

example : s ∩ (s ∪ t) = s :=
begin
  ext x,
  split,
  { intros h,
    dsimp at h,
    exact h.1, },
  { intro xs,
    dsimp,
    split,
    { exact xs, },
    { left,
      exact xs, }},
end

-- 2ª demostración
-- ===============

example : s ∩ (s ∪ t) = s :=
begin
  ext x,
  split,
  { intros h,
    exact h.1, },
  { intro xs,
    split,
    { exact xs, },
    { left,
```

```
      exact xs, }},
end

-- 3ª demostración
-- ===============

example : s ∩ (s ∪ t) = s :=
begin
  ext x,
  split,
  { intros h,
    exact h.1, },
  { intro xs,
    split,
    { exact xs, },
    { exact (or.inl xs), }},
end

-- 4ª demostración
-- ===============

example : s ∩ (s ∪ t) = s :=
begin
  ext,
  exact ⟨λ h, h.1,
         λ xs, ⟨xs, or.inl xs⟩⟩,
end

-- 5ª demostración
-- ===============

example : s ∩ (s ∪ t) = s :=
begin
  ext,
  exact ⟨and.left,
         λ xs, ⟨xs, or.inl xs⟩⟩,
end

-- 6ª demostración
-- ===============

example : s ∩ (s ∪ t) = s :=
begin
  ext x,
  split,
```

```
  { rintros ⟨xs, _⟩,
    exact xs },
  { intro xs,
    use xs,
    left,
    exact xs },
end

-- 7ª demostración
-- ===============

example : s ∩ (s ∪ t) = s :=
inf_sup_self
```

Se puede interactuar con las pruebas anteriores en esta sesión con Lean.

2.8. Unión con su intersección

```
-- ------------------------------------------------------------------------
-- Demostrar que
--    s ∪ (s ∩ t) = s
-- -------------------------------------------------------------------------

import data.set.basic
open set

variable {α : Type}
variables s t : set α

-- 1ª demostración
-- ===============

example : s ∪ (s ∩ t) = s :=
begin
  ext x,
  split,
  { intro hx,
    cases hx with xs xst,
    { exact xs, },
    { exact xst.1, }},
  { intro xs,
    left,
    exact xs, },
```

```
end

-- 2ª demostración
-- ===============

example : s ∪ (s ∩ t) = s :=
begin
  ext x,
  exact ⟨λ hx, or.dcases_on hx id and.left,
         λ xs, or.inl xs⟩,
end

-- 3ª demostración
-- ===============

example : s ∪ (s ∩ t) = s :=
begin
  ext x,
  split,
  { rintros (xs | ⟨xs, xt⟩);
    exact xs },
  { intro xs,
    left,
    exact xs },
end

-- 4ª demostración
-- ===============

example : s ∪ (s ∩ t) = s :=
sup_inf_self
```

Se puede interactuar con las pruebas anteriores en esta sesión con Lean.

2.9. Unión con su diferencia

```
-- ---------------------------------------------------------------------
-- Demostrar que
--    (s \ t) ∪ t = s ∪ t
-- ----------------------------------------------------------------------

import data.set.basic
open set
```

```
variable {α : Type}
variables s t : set α

-- 1ª definición
-- =============

example : (s \ t) ∪ t = s ∪ t :=
begin
  ext x,
  split,
  { intro hx,
    cases hx with xst xt,
    { left,
      exact xst.1, },
    { right,
      exact xt }},
  { by_cases h : x ∈ t,
    { intro _,
      right,
      exact h },
    { intro hx,
      cases hx with xs xt,
      { left,
        split,
        { exact xs, },
        { dsimp,
          exact h, }},
      { right,
        exact xt, }}},
end

-- 2ª definición
-- =============

example : (s \ t) ∪ t = s ∪ t :=
begin
  ext x,
  split,
  { rintros (⟨xs, nxt⟩ | xt),
    { left,
      exact xs},
    { right,
      exact xt }},
  { by_cases h : x ∈ t,
```

```
    { intro _,
      right,
      exact h },
    { rintros (xs | xt),
      { left,
        use [xs, h] },
      { right,
        use xt }}},
end

-- 3ª definición
-- =============

example : (s \ t) ∪ t = s ∪ t :=
begin
  rw ext_iff,
  intro,
  rw iff_def,
  finish,
end

-- 4ª definición
-- =============

example : (s \ t) ∪ t = s ∪ t :=
by finish [ext_iff, iff_def]

-- 5ª definición
-- =============

example : (s \ t) ∪ t = s ∪ t :=
diff_union_self

-- 6ª definición
-- =============

example : (s \ t) ∪ t = s ∪ t :=
begin
  ext,
  simp,
end

-- 7ª definición
-- =============
```

```
example : (s \ t) ∪ t = s ∪ t :=
by simp
```

Se puede interactuar con las pruebas anteriores en esta sesión con Lean.

2.10. Diferencia de unión e intersección

```
-- ----------------------------------------------------------------------
-- Demostrar que
--    (s \ t) ∪ (t \ s) = (s ∪ t) \ (s ∩ t)
-- -----------------------------------------------------------------------

import data.set.basic
open set

variable {α : Type}
variables s t : set α

-- 1ª demostración
-- ===============

example : (s \ t) ∪ (t \ s) = (s ∪ t) \ (s ∩ t) :=
begin
  ext x,
  split,
  { rintros (⟨xs, xnt⟩ | ⟨xt, xns⟩),
    { split,
      { left,
        exact xs },
      { rintros ⟨_, xt⟩,
        contradiction }},
    { split ,
      { right,
        exact xt },
      { rintros ⟨xs, _⟩,
        contradiction }}},
  { rintros ⟨xs | xt, nxst⟩,
    { left,
      use xs,
      intro xt,
      apply nxst,
      split; assumption },
    { right,
```

```
      use xt,
      intro xs,
      apply nxst,
      split; assumption }},
end

-- 2ª demostración
-- ===============

example : (s \ t) ∪ (t \ s) = (s ∪ t) \ (s ∩ t) :=
begin
  ext x,
  split,
  { rintros (⟨xs, xnt⟩ | ⟨xt, xns⟩),
    { finish, },
    { finish, }},
  { rintros ⟨xs | xt, nxst⟩,
    { finish, },
    { finish, }},
end

-- 3ª demostración
-- ===============

example : (s \ t) ∪ (t \ s) = (s ∪ t) \ (s ∩ t) :=
begin
  ext x,
  split,
  { rintros (⟨xs, xnt⟩ | ⟨xt, xns⟩) ; finish, },
  { rintros ⟨xs | xt, nxst⟩ ; finish, },
end

-- 4ª demostración
-- ===============

example : (s \ t) ∪ (t \ s) = (s ∪ t) \ (s ∩ t) :=
begin
  ext,
  split,
  { finish, },
  { finish, },
end

-- 5ª demostración
-- ===============
```

```
example : (s \ t) ∪ (t \ s) = (s ∪ t) \ (s ∩ t) :=
begin
  rw ext_iff,
  intro,
  rw iff_def,
  finish,
end

-- 6ª demostración
-- ===============

example : (s \ t) ∪ (t \ s) = (s ∪ t) \ (s ∩ t) :=
by finish [ext_iff, iff_def]
```

Se puede interactuar con las pruebas anteriores en esta sesión con Lean.

2.11. Unión de los conjuntos de los pares e impares

```
-- ----------------------------------------------------------------------
-- Los conjuntos de los números naturales, de los pares y de los impares
-- se definen por
--    def naturales : set ℕ := {n | true}
--    def pares     : set ℕ := {n | even n}
--    def impares   : set ℕ := {n | ¬ even n}
--
-- Demostrar que
--    pares ∪ impares = naturales
-- ----------------------------------------------------------------------

import data.nat.parity
import data.set.basic
import tactic

open set

def naturales : set ℕ := {n | true}
def pares     : set ℕ := {n | even n}
def impares   : set ℕ := {n | ¬ even n}

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

```
example : pares ∪ impares = naturales :=
begin
  unfold pares impares naturales,
  ext n,
  simp,
  apply classical.em,
end

-- 2ª demostración
-- ===============

example : pares ∪ impares = naturales :=
begin
  unfold pares impares naturales,
  ext n,
  finish,
end

-- 3ª demostración
-- ===============

example : pares ∪ impares = naturales :=
by finish [pares, impares, naturales, ext_iff]
```

Se puede interactuar con las pruebas anteriores en esta sesión con Lean.

Capítulo 3

Ejercicios de junio de 2021

3.1. Intersección de los primos y los mayores que dos

```
-- ---------------------------------------------------------------------
-- Los números primos, los mayores que 2 y los impares se definen por
--    def primos      : set ℕ := {n | prime n}
--    def mayoresQue2 : set ℕ := {n | n > 2}
--    def impares     : set ℕ := {n | ¬ even n}
--
-- Demostrar que
--    primos ∩ mayoresQue2 ⊆ impares
-- ----------------------------------------------------------------------

import data.nat.parity
import data.nat.prime
import tactic

open nat

def primos      : set ℕ := {n | prime n}
def mayoresQue2 : set ℕ := {n | n > 2}
def impares     : set ℕ := {n | ¬ even n}

example : primos ∩ mayoresQue2 ⊆ impares :=
begin
  unfold primos mayoresQue2 impares,
  intro n,
  simp,
  intro hn,
```

```
  cases prime.eq_two_or_odd hn with h h,
  { rw h,
    intro,
    linarith, },
  { rw even_iff,
    rw h,
    norm_num },
end
```

Se puede interactuar con las pruebas anteriores en esta sesión con Lean.

3.2. Distributiva de la intersección respecto de la unión general

```
-- ----------------------------------------------------------------------
-- Demostrar que
--    s ∩ (⋃ i, A i) = ⋃ i, (A i ∩ s)
-- -----------------------------------------------------------------------

import data.set.basic
import data.set.lattice
import tactic

open set

variable {α : Type}
variable s : set α
variable A : ℕ → set α

-- 1ª demostración
-- ===============

example : s ∩ (⋃ i, A i) = ⋃ i, (A i ∩ s) :=
begin
  ext x,
  split,
  { intro h,
    rw mem_Union,
    cases h with xs xUAi,
    rw mem_Union at xUAi,
    cases xUAi with i xAi,
    use i,
```

```
    split,
    { exact xAi, },
    { exact xs, }},
  { intro h,
    rw mem_Union at h,
    cases h with i hi,
    cases hi with xAi xs,
    split,
    { exact xs, },
    { rw mem_Union,
      use i,
      exact xAi, }},
end

-- 2ª demostración
-- ===============

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

-- 3ª demostración
-- ===============

example : s ∩ (⋃ i, A i) = ⋃ i, (A i ∩ s) :=
begin
  ext x,
  finish [mem_inter_eq, mem_Union],
end

-- 4ª demostración
-- ===============

example : s ∩ (⋃ i, A i) = ⋃ i, (A i ∩ s) :=
by finish [mem_inter_eq, mem_Union, ext_iff]
```

Se puede interactuar con las pruebas anteriores en esta sesión con Lean.

3.3. Intersección de intersecciones

```
-- ----------------------------------------------------------------------
-- Demostrar que
--    (⋂ i, A i ∩ B i) = (⋂ i, A i) ∩ (⋂ i, B i)
-- -----------------------------------------------------------------------

import data.set.basic
import tactic

open set

variable  {α : Type}
variables A B : ℕ → set α

-- 1ª demostración
-- ===============

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 }},
  { intros h i,
    cases h with h1 h2,
    split,
    { exact h1 i },
    { exact h2 i }},
end

-- 2ª demostración
-- ===============

example : (⋂ i, A i ∩ B i) = (⋂ i, A i) ∩ (⋂ i, B i) :=
begin
  ext x,
  simp only [mem_inter_eq, mem_Inter],
  exact ⟨λ h, ⟨λ i, (h i).1, λ i, (h i).2⟩,
```

```
        λ ⟨h1, h2⟩ i, ⟨h1 i, h2 i⟩⟩,
end

-- 3ª demostración
-- ===============

example : (⋂ i, A i ∩ B i) = (⋂ i, A i) ∩ (⋂ i, B i) :=
begin
  ext,
  simp only [mem_inter_eq, mem_Inter],
  finish,
end

-- 4ª demostración
-- ===============

example : (⋂ i, A i ∩ B i) = (⋂ i, A i) ∩ (⋂ i, B i) :=
begin
  ext,
  finish [mem_inter_eq, mem_Inter],
end

-- 5ª demostración
-- ===============

example : (⋂ i, A i ∩ B i) = (⋂ i, A i) ∩ (⋂ i, B i) :=
by finish [mem_inter_eq, mem_Inter, ext_iff]
```

Se puede interactuar con las pruebas anteriores en esta sesión con Lean.

3.4. Unión con intersección general

```
-- ----------------------------------------------------------------------
-- Demostrar que
--    s ∪ (⋂ i, A i) = ⋂ i, (A i ∪ s)
-- -----------------------------------------------------------------------

import data.set.basic
import tactic

open set

variable  {α : Type}
```

```
variable  s : set α
variables A : ℕ → set α

-- 1ª demostración
-- ===============

example : s ∪ (⋂ i, A i) = ⋂ i, (A i ∪ s) :=
begin
  ext x,
  simp only [mem_union, mem_Inter],
  split,
  { intros h i,
    cases h with xs xAi,
    { right,
      exact xs },
    { left,
      exact xAi i, }},
  { intro h,
    by_cases xs : x ∈ s,
    { left,
      exact xs },
    { right,
      intro i,
      cases h i with xAi xs,
      { exact xAi, },
      { contradiction, }}},
end

-- 2ª demostración
-- ===============

example : s ∪ (⋂ i, A i) = ⋂ i, (A i ∪ s) :=
begin
  ext x,
  simp only [mem_union, mem_Inter],
  split,
  { rintros (xs | xI) i,
    { right,
      exact xs },
    { left,
      exact xI i }},
  { intro h,
    by_cases xs : x ∈ s,
    { left,
      exact xs },
```

```
    { right,
      intro i,
      cases h i,
      { assumption },
      { contradiction }}},
end

-- 3ª demostración
-- ===============

example : s ∪ (⋂ i, A i) = ⋂ i, (A i ∪ s) :=
begin
  ext x,
  simp only [mem_union, mem_Inter],
  split,
  { finish, },
  { finish, },
end

-- 4ª demostración
-- ===============

example : s ∪ (⋂ i, A i) = ⋂ i, (A i ∪ s) :=
begin
  ext,
  simp only [mem_union, mem_Inter],
  split ; finish,
end

-- 5ª demostración
-- ===============

example : s ∪ (⋂ i, A i) = ⋂ i, (A i ∪ s) :=
begin
  ext,
  simp only [mem_union, mem_Inter],
  finish [iff_def],
end

-- 6ª demostración
-- ===============

example : s ∪ (⋂ i, A i) = ⋂ i, (A i ∪ s) :=
by finish [ext_iff, mem_union, mem_Inter, iff_def]
```

Se puede interactuar con las pruebas anteriores en esta sesión con Lean.

3.5. Imagen inversa de la intersección

```
-- ---------------------------------------------------------------------
-- En Lean, la imagen inversa de un conjunto s (de elementos de tipo β)
-- por la función f (de tipo α → β) es el conjunto `f ⁻¹' s` de
-- elementos x (de tipo α) tales que `f x ∈ s`.
--
-- Demostrar que
--    f ⁻¹' (u ∩ v) = f ⁻¹' u ∩ f ⁻¹' v
-- ----------------------------------------------------------------------

import data.set.basic

open set

variables {α : Type*} {β : Type*}
variable  f : α → β
variables u v : set β

-- 1ª demostración
-- ===============

example : f ⁻¹' (u ∩ v) = f ⁻¹' u ∩ f ⁻¹' v :=
begin
  ext x,
  split,
  { intro h,
    split,
    { apply mem_preimage.mpr,
      rw mem_preimage at h,
      exact mem_of_mem_inter_left h, },
    { apply mem_preimage.mpr,
      rw mem_preimage at h,
      exact mem_of_mem_inter_right h, }},
  { intro h,
    apply mem_preimage.mpr,
    split,
    { apply mem_preimage.mp,
      exact mem_of_mem_inter_left h,},
    { apply mem_preimage.mp,
      exact mem_of_mem_inter_right h, }},
```

```
end

-- 2ª demostración
-- ===============

example : f ⁻¹' (u ∩ v) = f ⁻¹' u ∩ f ⁻¹' v :=
begin
  ext x,
  exact ⟨λ h, ⟨mem_preimage.mpr (mem_of_mem_inter_left h),
               mem_preimage.mpr (mem_of_mem_inter_right h)⟩,
         λ h, ⟨mem_preimage.mp (mem_of_mem_inter_left h),
               mem_preimage.mp (mem_of_mem_inter_right h)⟩⟩,
end

-- 3ª demostración
-- ===============

example : f ⁻¹' (u ∩ v) = f ⁻¹' u ∩ f ⁻¹' v :=
begin
  ext,
  refl,
end

-- 4ª demostración
-- ===============

example : f ⁻¹' (u ∩ v) = f ⁻¹' u ∩ f ⁻¹' v :=
by {ext, refl}

-- 5ª demostración
-- ===============

example : f ⁻¹' (u ∩ v) = f ⁻¹' u ∩ f ⁻¹' v :=
rfl

-- 6ª demostración
-- ===============

example : f ⁻¹' (u ∩ v) = f ⁻¹' u ∩ f ⁻¹' v :=
preimage_inter
```

Se puede interactuar con las pruebas anteriores en esta sesión con Lean.

3.6. Imagen de la unión

```
-- ---------------------------------------------------------------------
-- En Lean, la imagen de un conjunto s por una función f se representa
-- por `f '' s`; es decir,
--    f '' s = {y | ∃ x, x ∈ s ∧ f x = y}
--
-- Demostrar que
--    f '' (s ∪ t) = f '' s ∪ f '' t
-- ----------------------------------------------------------------------

import data.set.basic
import tactic

open set

variables {α : Type*} {β : Type*}
variable  f : α → β
variables s t : set α

-- 1ª demostración
-- ===============

example : f '' (s ∪ t) = f '' s ∪ f '' t :=
begin
  ext y,
  split,
  { intro h1,
    cases h1 with x hx,
    cases hx with xst fxy,
    rw ← fxy,
    cases xst with xs xt,
    { left,
      apply mem_image_of_mem,
      exact xs, },
    { right,
      apply mem_image_of_mem,
      exact xt, }},
  { intro h2,
    cases h2 with yfs yft,
    { cases yfs with x hx,
      cases hx with xs fxy,
      rw ← fxy,
      apply mem_image_of_mem,
```

```
      left,
      exact xs, },
    { cases yft with x hx,
      cases hx with xt fxy,
      rw ← fxy,
      apply mem_image_of_mem,
      right,
      exact xt, }},
end

-- 2ª demostración
-- ===============

example : f '' (s ∪ t) = f '' s ∪ f '' t :=
begin
  ext y,
  split,
  { rintro ⟨x, xst, fxy⟩,
    rw ← fxy,
    cases xst with xs xt,
    { left,
      exact mem_image_of_mem f xs, },
    { right,
      exact mem_image_of_mem f xt, }},
  { rintros (yfs | yft),
    { rcases yfs with ⟨x, xs, fxy⟩,
      rw ← fxy,
      apply mem_image_of_mem,
      left,
      exact xs, },
    { rcases yft with ⟨x, xt, fxy⟩,
      rw ← fxy,
      apply mem_image_of_mem,
      right,
      exact xt, }},
end

-- 3ª demostración
-- ===============

example : f '' (s ∪ t) = f '' s ∪ f '' t :=
begin
  ext y,
  split,
  { rintro ⟨x, xst, rfl⟩,
```

```
    cases xst with xs xt,
    { left,
      exact mem_image_of_mem f xs, },
    { right,
      exact mem_image_of_mem f xt, }},
  { rintros (yfs | yft),
    { rcases yfs with ⟨x, xs, rfl⟩,
      apply mem_image_of_mem,
      left,
      exact xs, },
    { rcases yft with ⟨x, xt, rfl⟩,
      apply mem_image_of_mem,
      right,
      exact xt, }},
end

-- 4ª demostración
-- ===============

example : f '' (s ∪ t) = f '' s ∪ f '' t :=
begin
  ext y,
  split,
  { rintro ⟨x, xst, rfl⟩,
    cases xst with xs xt,
    { left,
      use [x, xs], },
    { right,
      use [x, xt], }},
  { rintros (yfs | yft),
    { rcases yfs with ⟨x, xs, rfl⟩,
      use [x, or.inl xs], },
    { rcases yft with ⟨x, xt, rfl⟩,
      use [x, or.inr xt], }},
end

-- 5ª demostración
-- ===============

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

-- 6ª demostración
-- ===============

example : f '' (s ∪ t) = f '' s ∪ f '' t :=
begin
  ext y,
  split,
  { rintros ⟨x, xs | xt, rfl⟩,
    { finish, },
    { finish, }},
  { rintros (⟨x, xs, rfl⟩ | ⟨x, xt, rfl⟩),
    { finish, },
    { finish, }},
end

-- 7ª demostración
-- ===============

example : f '' (s ∪ t) = f '' s ∪ f '' t :=
begin
  ext y,
  split,
  { rintros ⟨x, xs | xt, rfl⟩ ; finish, },
  { rintros (⟨x, xs, rfl⟩ | ⟨x, xt, rfl⟩) ; finish, },
end

-- 8ª demostración
-- ===============

example : f '' (s ∪ t) = f '' s ∪ f '' t :=
begin
  ext y,
  split,
  { finish, },
  { finish, },
end
```

```
-- 9ª demostración
-- ===============

example : f '' (s ∪ t) = f '' s ∪ f '' t :=
begin
  ext y,
  rw iff_def,
  finish,
end

-- 10ª demostración
-- ===============

example : f '' (s ∪ t) = f '' s ∪ f '' t :=
by finish [ext_iff, iff_def, mem_image_eq]

-- 11ª demostración
-- ===============

example : f '' (s ∪ t) = f '' s ∪ f '' t :=
image_union f s t
```

Se puede interactuar con las pruebas anteriores en esta sesión con Lean.

3.7. Imagen inversa de la imagen

```
-- ---------------------------------------------------------------------
-- Demostrar que si s es un subconjunto del dominio de la función f,
-- entonces s está contenido en la [imagen inversa](https://bit.ly/3ckseBL)
-- de la [imagen de s por f](https://bit.ly/3x2Jxij); es decir,
--    s ⊆ f⁻¹[f[s]]
-- ----------------------------------------------------------------------

import data.set.basic

open set

variables {α : Type*} {β : Type*}
variable  f : α → β
variable  s : set α

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

```
example : s ⊆ f ⁻¹' (f '' s) :=
begin
  intros x xs,
  apply mem_preimage.mpr,
  apply mem_image_of_mem,
  exact xs,
end

-- 2ª demostración
-- ===============

example : s ⊆ f ⁻¹' (f '' s) :=
begin
  intros x xs,
  apply mem_image_of_mem,
  exact xs,
end

-- 3ª demostración
-- ===============

example : s ⊆ f ⁻¹' (f '' s) :=
λ x, mem_image_of_mem f

-- 4ª demostración
-- ===============

example : s ⊆ f ⁻¹' (f '' s) :=
begin
  intros x xs,
  show f x ∈ f '' s,
  use [x, xs],
end

-- 5ª demostración
-- ===============

example : s ⊆ f ⁻¹' (f '' s) :=
begin
  intros x xs,
  use [x, xs],
end

-- 6ª demostración
```

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

example : s ⊆ f ⁻¹' (f '' s) :=
subset_preimage_image f s
```

Se puede interactuar con las pruebas anteriores en esta sesión con Lean.

3.8. Subconjunto de la imagen inversa

```
-- ----------------------------------------------------------------------
-- Demostrar que
--    f[s] ⊆ u ↔ s ⊆ f⁻¹[u]
-- ----------------------------------------------------------------------

import data.set.basic

open set

variables {α : Type*} {β : Type*}
variable  f : α → β
variable  s : set α
variable  u : set β

-- 1ª demostración
-- ===============

example : f '' s ⊆ u ↔ s ⊆ f ⁻¹' u :=
begin
  split,
  { intros h x xs,
    apply mem_preimage.mpr,
    apply h,
    apply mem_image_of_mem,
    exact xs, },
  { intros h y hy,
    rcases hy with ⟨x, xs, fxy⟩,
    rw ← fxy,
    exact h xs, },
end

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

```
example : f '' s ⊆ u ↔ s ⊆ f ⁻¹' u :=
begin
  split,
  { intros h x xs,
    apply h,
    apply mem_image_of_mem,
    exact xs, },
  { rintros h y ⟨x, xs, rfl⟩,
    exact h xs, },
end

-- 3ª demostración
-- ===============

example : f '' s ⊆ u ↔ s ⊆ f ⁻¹' u :=
image_subset_iff

-- 4ª demostración
-- ===============

example : f '' s ⊆ u ↔ s ⊆ f ⁻¹' u :=
by simp
```

Se puede interactuar con las pruebas anteriores en esta sesión con Lean.

3.9. Imagen inversa de la imagen de aplicaciones inyectivas

```
-- ----------------------------------------------------------------------
-- Demostrar que si f es inyectiva, entonces
--    f⁻¹[f[s]] ⊆ s
-- -----------------------------------------------------------------------

import data.set.basic

open set function

variables {α : Type*} {β : Type*}
variable  f : α → β
variable  s : set α

-- 1ª demostración
```

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

example
  (h : injective f)
  : f ⁻¹' (f '' s) ⊆ s :=
begin
  intros x hx,
  rw mem_preimage at hx,
  rw mem_image_eq at hx,
  cases hx with y hy,
  cases hy with ys fyx,
  unfold injective at h,
  have h1 : y = x := h fyx,
  rw ← h1,
  exact ys,
end

-- 2ª demostración
-- ===============

example
  (h : injective f)
  : f ⁻¹' (f '' s) ⊆ s :=
begin
  intros x hx,
  rw mem_preimage at hx,
  rcases hx with ⟨y, ys, fyx⟩,
  rw ← h fyx,
  exact ys,
end

-- 3ª demostración
-- ===============

example
  (h : injective f)
  : f ⁻¹' (f '' s) ⊆ s :=
begin
  rintros x ⟨y, ys, hy⟩,
  rw ← h hy,
  exact ys,
end
```

Se puede interactuar con las pruebas anteriores en esta sesión con Lean.

3.10. Imagen de la imagen inversa

```
-- ----------------------------------------------------------------------
-- Demostrar que
--    f '' (f⁻¹' u) ⊆ u
-- ----------------------------------------------------------------------

import data.set.basic
open set

variables {α : Type*} {β : Type*}
variable  f : α → β
variable  u : set β

-- 1ª demostración
-- ===============

example : f '' (f⁻¹' u) ⊆ u :=
begin
  intros y h,
  cases h with x h2,
  cases h2 with hx fxy,
  rw ← fxy,
  exact hx,
end

-- 2ª demostración
-- ===============

example : f '' (f⁻¹' u) ⊆ u :=
begin
  intros y h,
  rcases h with ⟨x, hx, fxy⟩,
  rw ← fxy,
  exact hx,
end

-- 3ª demostración
-- ===============

example : f '' (f⁻¹' u) ⊆ u :=
begin
  rintros y ⟨x, hx, fxy⟩,
  rw ← fxy,
```

```
  exact hx,
end

-- 4ª demostración
-- ===============

example : f '' (f⁻¹' u) ⊆ u :=
begin
  rintros y ⟨x, hx, rfl⟩,
  exact hx,
end

-- 5ª demostración
-- ===============

example : f '' (f⁻¹' u) ⊆ u :=
image_preimage_subset f u

-- 6ª demostración
-- ===============

example : f '' (f⁻¹' u) ⊆ u :=
by simp
```

Se puede interactuar con las pruebas anteriores en esta sesión con Lean.

3.11. Imagen de imagen inversa de aplicaciones suprayectivas

```
-- ----------------------------------------------------------------------
-- Demostrar que si f es suprayectiva, entonces
--    u ⊆ f '' (f⁻¹' u)
-- ----------------------------------------------------------------------

import data.set.basic
open set function

variables {α : Type*} {β : Type*}
variable  f : α → β
variable  u : set β

-- 1ª demostración
```

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

example
  (h : surjective f)
  : u ⊆ f '' (f⁻¹' u) :=
begin
  intros y yu,
  cases h y with x fxy,
  use x,
  split,
  { apply mem_preimage.mpr,
    rw fxy,
    exact yu },
  { exact fxy },
end

-- 2ª demostración
-- ===============

example
  (h : surjective f)
  : u ⊆ f '' (f⁻¹' u) :=
begin
  intros y yu,
  cases h y with x fxy,
  use x,
  split,
  { show f x ∈ u,
    rw fxy,
    exact yu },
  { exact fxy },
end

-- 3ª demostración
-- ===============

example
  (h : surjective f)
  : u ⊆ f '' (f⁻¹' u) :=
begin
  intros y yu,
  cases h y with x fxy,
  by finish,
end
```

Se puede interactuar con las pruebas anteriores en esta sesión con Lean.

3.12. Monotonía de la imagen de conjuntos

```
-- ---------------------------------------------------------------------
-- Demostrar que si s ⊆ t, entonces
--    f '' s ⊆ f '' t
-- ----------------------------------------------------------------------

import data.set.basic
import tactic

open set

variables {α : Type*} {β : Type*}
variable  f : α → β
variables s t : set α

-- 1ª demostración
-- ===============

example
  (h : s ⊆ t)
  : f '' s ⊆ f '' t :=
begin
  intros y hy,
  rw mem_image at hy,
  cases hy with x hx,
  cases hx with xs fxy,
  use x,
  split,
  { exact h xs, },
  { exact fxy, },
end

-- 2ª demostración
-- ===============

example
  (h : s ⊆ t)
  : f '' s ⊆ f '' t :=
begin
  intros y hy,
  rcases hy with ⟨x, xs, fxy⟩,
  use x,
  exact ⟨h xs, fxy⟩,
```

```
end

-- 3ª demostración
-- ===============

example
  (h : s ⊆ t)
  : f '' s ⊆ f '' t :=
begin
  rintros y ⟨x, xs, fxy ⟩,
  use [x, h xs, fxy],
end

-- 4ª demostración
-- ===============

example
  (h : s ⊆ t)
  : f '' s ⊆ f '' t :=
by finish [subset_def, mem_image_eq]

-- 5ª demostración
-- ===============

example
  (h : s ⊆ t)
  : f '' s ⊆ f '' t :=
image_subset f h
```

Se puede interactuar con las pruebas anteriores en esta sesión con Lean.

3.13. Monotonía de la imagen inversa

```
-- ----------------------------------------------------------------------
-- Demostrar que si u ⊆ v, entonces
--    f ⁻¹' u ⊆ f ⁻¹' v
-- -----------------------------------------------------------------------

import data.set.basic
open set

variables {α : Type*} {β : Type*}
variable  f : α → β
```

```
variables u v : set β

-- 1ª demostración
-- ===============

example
  (h : u ⊆ v)
  : f ⁻¹' u ⊆ f ⁻¹' v :=
begin
  intros x hx,
  apply mem_preimage.mpr,
  apply h,
  apply mem_preimage.mp,
  exact hx,
end

-- 2ª demostración
-- ===============

example
  (h : u ⊆ v)
  : f ⁻¹' u ⊆ f ⁻¹' v :=
begin
  intros x hx,
  apply h,
  exact hx,
end

-- 3ª demostración
-- ===============

example
  (h : u ⊆ v)
  : f ⁻¹' u ⊆ f ⁻¹' v :=
begin
  intros x hx,
  exact h hx,
end

-- 4ª demostración
-- ===============

example
  (h : u ⊆ v)
```

```
  : f ⁻¹' u ⊆ f ⁻¹' v :=
λ x hx, h hx

-- 5ª demostración
-- ===============

example
  (h : u ⊆ v)
  : f ⁻¹' u ⊆ f ⁻¹' v :=
by intro x; apply h

-- 6ª demostración
-- ===============

example
  (h : u ⊆ v)
  : f ⁻¹' u ⊆ f ⁻¹' v :=
preimage_mono h

-- 7ª demostración
-- ===============

example
  (h : u ⊆ v)
  : f ⁻¹' u ⊆ f ⁻¹' v :=
by tauto
```

Se puede interactuar con las pruebas anteriores en esta sesión con Lean.

3.14. Imagen inversa de la unión

```
-- ----------------------------------------------------------------------
-- Demostrar que
--    f ⁻¹' (u ∪ v) = f ⁻¹' u ∪ f ⁻¹' v
-- ----------------------------------------------------------------------

import data.set.basic

open set

variables {α : Type*} {β : Type*}
variable  f : α → β
```

```
variables u v : set β

-- 1ª demostración
-- ===============

example : f ⁻¹' (u ∪ v) = f ⁻¹' u ∪ f ⁻¹' v :=
begin
  ext x,
  split,
  { intros h,
    rw mem_preimage at h,
    cases h with fxu fxv,
    { left,
      apply mem_preimage.mpr,
      exact fxu, },
    { right,
      apply mem_preimage.mpr,
      exact fxv, }},
  { intro h,
    rw mem_preimage,
    cases h with xfu xfv,
    { rw mem_preimage at xfu,
      left,
      exact xfu, },
    { rw mem_preimage at xfv,
      right,
      exact xfv, }},
end

-- 2ª demostración
-- ===============

example : f ⁻¹' (u ∪ v) = f ⁻¹' u ∪ f ⁻¹' v :=
begin
  ext x,
  split,
  { intros h,
    cases h with fxu fxv,
    { left,
      exact fxu, },
    { right,
      exact fxv, }},
  { intro h,
    cases h with xfu xfv,
    { left,
```

```
      exact xfu, },
    { right,
      exact xfv, }},
end

-- 3ª demostración
-- ===============

example : f ⁻¹' (u ∪ v) = f ⁻¹' u ∪ f ⁻¹' v :=
begin
  ext x,
  split,
  { rintro (fxu | fxv),
    { exact or.inl fxu, },
    { exact or.inr fxv, }},
  { rintro (xfu | xfv),
    { exact or.inl xfu, },
    { exact or.inr xfv, }},
end

-- 4ª demostración
-- ===============

example : f ⁻¹' (u ∪ v) = f ⁻¹' u ∪ f ⁻¹' v :=
begin
  ext x,
  split,
  { finish, },
  { finish, } ,
end

-- 5ª demostración
-- ===============

example : f ⁻¹' (u ∪ v) = f ⁻¹' u ∪ f ⁻¹' v :=
begin
  ext x,
  finish,
end

-- 6ª demostración
-- ===============

example : f ⁻¹' (u ∪ v) = f ⁻¹' u ∪ f ⁻¹' v :=
by ext; finish
```

```
-- 7ª demostración
-- ===============

example : f ⁻¹' (u ∪ v) = f ⁻¹' u ∪ f ⁻¹' v :=
by ext; refl

-- 8ª demostración
-- ===============

example : f ⁻¹' (u ∪ v) = f ⁻¹' u ∪ f ⁻¹' v :=
rfl

-- 9ª demostración
-- ===============

example : f ⁻¹' (u ∪ v) = f ⁻¹' u ∪ f ⁻¹' v :=
preimage_union

-- 10ª demostración
-- ===============

example : f ⁻¹' (u ∪ v) = f ⁻¹' u ∪ f ⁻¹' v :=
by simp
```

Se puede interactuar con las pruebas anteriores en esta sesión con Lean.

3.15. Imagen de la intersección

```
-- ----------------------------------------------------------------------
-- Demostrar que
--    f '' (s ∩ t) ⊆ f '' s ∩ f '' t
-- -----------------------------------------------------------------------

import data.set.basic
import tactic

open set

variables {α : Type*} {β : Type*}
variable  f : α → β
variables s t : set α

-- 1ª demostración
```

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

example : f '' (s ∩ t) ⊆ f '' s ∩ f '' t :=
begin
  intros y hy,
  cases hy with x hx,
  cases hx with xst fxy,
  split,
  { use x,
    split,
    { exact xst.1, },
    { exact fxy, }},
  { use x,
    split,
    { exact xst.2, },
    { exact fxy, }},
end

-- 2ª demostración
-- ===============

example : f '' (s ∩ t) ⊆ f '' s ∩ f '' t :=
begin
  intros y hy,
  rcases hy with ⟨x, ⟨xs, xt⟩, fxy⟩,
  split,
  { use x,
    exact ⟨xs, fxy⟩, },
  { use x,
    exact ⟨xt, fxy⟩, },
end

-- 3ª demostración
-- ===============

example : f '' (s ∩ t) ⊆ f '' s ∩ f '' t :=
begin
  rintros y ⟨x, ⟨xs, xt⟩, fxy⟩,
  split,
  { use [x, xs, fxy], },
  { use [x, xt, fxy], },
end

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

```
example : f '' (s ∩ t) ⊆ f '' s ∩ f '' t :=
image_inter_subset f s t

-- 5ª demostración
-- ===============

example : f '' (s ∩ t) ⊆ f '' s ∩ f '' t :=
by intro ; finish
```

Se puede interactuar con las pruebas anteriores en esta sesión con Lean.

3.16. Imagen de la intersección de aplicaciones inyectivas

```
-- ---------------------------------------------------------------------
-- Demostrar que si f es inyectiva, entonces
--    f '' s ∩ f '' t ⊆ f '' (s ∩ t)
-- ----------------------------------------------------------------------

import data.set.basic

open set function

variables {α : Type*} {β : Type*}
variable  f : α → β
variables s t : set α

-- 1ª demostración
-- ===============

example
  (h : injective f)
  : f '' s ∩ f '' t ⊆ f '' (s ∩ t) :=
begin
  intros y hy,
  cases hy  with hy1 hy2,
  cases hy1 with x1 hx1,
  cases hx1 with x1s fx1y,
  cases hy2 with x2 hx2,
  cases hx2 with x2t fx2y,
  use x1,
  split,
```

```
  { split,
    { exact x1s, },
    { convert x2t,
      apply h,
      rw ← fx2y at fx1y,
      exact fx1y, }},
  { exact fx1y, },
end

-- 2ª demostración
-- ===============

example
  (h : injective f)
  : f '' s ∩ f '' t ⊆ f '' (s ∩ t) :=
begin
  rintros y ⟨⟨x1, x1s, fx1y⟩, ⟨x2, x2t, fx2y⟩⟩,
  use x1,
  split,
  { split,
    { exact x1s, },
    { convert x2t,
      apply h,
      rw ← fx2y at fx1y,
      exact fx1y, }},
  { exact fx1y, },
end

-- 3ª demostración
-- ===============

example
  (h : injective f)
  : f '' s ∩ f '' t ⊆ f '' (s ∩ t) :=
begin
  rintros y ⟨⟨x1, x1s, fx1y⟩, ⟨x2, x2t, fx2y⟩⟩,
  unfold injective at h,
  finish,
end

-- 4ª demostración
-- ===============

example
  (h : injective f)
```

```
  : f '' s ∩ f '' t ⊆ f '' (s ∩ t) :=
by intro ; unfold injective at *  ; finish
```

Se puede interactuar con las pruebas anteriores en esta sesión con Lean.

3.17. Imagen de la diferencia de conjuntos

```
-- ---------------------------------------------------------------------
-- Demostrar que
--    f '' s \ f '' t ⊆ f '' (s \ t)
-- ----------------------------------------------------------------------

import data.set.basic
import tactic

open set

variables {α : Type*} {β : Type*}
variable  f : α → β
variables s t : set α

-- 1ª demostración
-- ===============

example : f '' s \ f '' t ⊆ f '' (s \ t) :=
begin
  intros y hy,
  cases hy with yfs ynft,
  cases yfs with x hx,
  cases hx with xs fxy,
  use x,
  split,
  { split,
    { exact xs, },
    { dsimp,
      intro xt,
      apply ynft,
      rw ← fxy,
      apply mem_image_of_mem,
      exact xt, }},
  { exact fxy, },
end
```

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

example : f '' s \ f '' t ⊆ f '' (s \ t) :=
begin
  rintros y ⟨⟨x, xs, fxy⟩, ynft⟩,
  use x,
  split,
  { split,
    { exact xs, },
    { intro xt,
      apply ynft,
      use [x, xt, fxy], }},
  { exact fxy, },
end

-- 3ª demostración
-- ===============

example : f '' s \ f '' t ⊆ f '' (s \ t) :=
begin
  rintros y ⟨⟨x, xs, fxy⟩, ynft⟩,
  use x,
  finish,
end

-- 4ª demostración
-- ===============

example : f '' s \ f '' t ⊆ f '' (s \ t) :=
subset_image_diff f s t
```

Se puede interactuar con las pruebas anteriores en esta sesión con Lean.

3.18. Imagen inversa de la diferencia

```
-- ----------------------------------------------------------------------
-- Demostrar que
--    f ⁻¹' u \ f ⁻¹' v ⊆ f ⁻¹' (u \ v)
-- -----------------------------------------------------------------------

import data.set.basic
```

```
open set

variables {α : Type*} {β : Type*}
variable  f : α → β
variables u v : set β

-- 1ª demostración
-- ===============

example : f ⁻¹' u \ f ⁻¹' v ⊆ f ⁻¹' (u \ v) :=
begin
  intros x hx,
  rw mem_preimage,
  split,
  { rw ← mem_preimage,
    exact hx.1, },
  { dsimp,
    rw ← mem_preimage,
    exact hx.2, },
end

-- 2ª demostración
-- ===============

example : f ⁻¹' u \ f ⁻¹' v ⊆ f ⁻¹' (u \ v) :=
begin
  intros x hx,
  split,
  { exact hx.1, },
  { exact hx.2, },
end

-- 3ª demostración
-- ===============

example : f ⁻¹' u \ f ⁻¹' v ⊆ f ⁻¹' (u \ v) :=
begin
  intros x hx,
  exact ⟨hx.1, hx.2⟩,
end

-- 4ª demostración
-- ===============

example : f ⁻¹' u \ f ⁻¹' v ⊆ f ⁻¹' (u \ v) :=
```

```
begin
  rintros x ⟨h1, h2⟩,
  exact ⟨h1, h2⟩,
end

-- 5ª demostración
-- ===============

example : f ⁻¹' u \ f ⁻¹' v ⊆ f ⁻¹' (u \ v) :=
subset.rfl

-- 6ª demostración
-- ===============

example : f ⁻¹' u \ f ⁻¹' v ⊆ f ⁻¹' (u \ v) :=
by finish
```

Se puede interactuar con las pruebas anteriores en esta sesión con Lean.

3.19. Intersección con la imagen

```
-- ---------------------------------------------------------------------
-- Demostrar que
--    (f '' s) ∩ v = f '' (s ∩ f ⁻¹' v)
-- ----------------------------------------------------------------------

import data.set.basic
import tactic

open set

variables {α : Type*} {β : Type*}
variable  f : α → β
variable  s : set α
variable  v : set β

-- 1ª demostración
-- ===============

example : (f '' s) ∩ v = f '' (s ∩ f ⁻¹' v) :=
begin
  ext y,
  split,
```

```
  { intro hy,
    cases hy with hyfs yv,
    cases hyfs with x hx,
    cases hx with xs fxy,
    use x,
    split,
    { split,
      { exact xs, },
      { rw mem_preimage,
        rw fxy,
        exact yv, }},
    { exact fxy, }},
  { intro hy,
    cases hy with x hx,
    split,
    { use x,
      split,
      { exact hx.1.1, },
      { exact hx.2, }},
    { cases hx with hx1 fxy,
      rw ← fxy,
      rw ← mem_preimage,
      exact hx1.2, }},
end

-- 2ª demostración
-- ===============

example : (f '' s) ∩ v = f '' (s ∩ f ⁻¹' v) :=
begin
  ext y,
  split,
  { rintros ⟨⟨x, xs, fxy⟩, yv⟩,
    use x,
    split,
    { split,
      { exact xs, },
      { rw mem_preimage,
        rw fxy,
        exact yv, }},
    { exact fxy, }},
  { rintros ⟨x, ⟨xs, xv⟩, fxy⟩,
    split,
    { use [x, xs, fxy], },
    { rw ← fxy,
```

```
      rw ← mem_preimage,
      exact xv, }},
end

-- 3ª demostración
-- ===============

example : (f '' s) ∩ v = f '' (s ∩ f ⁻¹' v) :=
begin
  ext y,
  split,
  { rintros ⟨⟨x, xs, fxy⟩, yv⟩,
    finish, },
  { rintros ⟨x, ⟨xs, xv⟩, fxy⟩,
    finish, },
end

-- 4ª demostración
-- ===============

example : (f '' s) ∩ v = f '' (s ∩ f ⁻¹' v) :=
by ext ; split ; finish

-- 5ª demostración
-- ===============

example : (f '' s) ∩ v = f '' (s ∩ f ⁻¹' v) :=
by finish [ext_iff, iff_def]

-- 6ª demostración
-- ===============

example : (f '' s) ∩ v = f '' (s ∩ f ⁻¹' v) :=
(image_inter_preimage f s v).symm
```

Se puede interactuar con las pruebas anteriores en esta sesión con Lean.

3.20. Unión con la imagen

```
-- ----------------------------------------------------------------------
-- Demostrar que
--    f '' (s ∪ f ⁻¹' v) ⊆ f '' s ∪ v
-- ----------------------------------------------------------------------
```

```
import data.set.basic
import tactic

open set

variables {α : Type*} {β : Type*}
variable  f : α → β
variable  s : set α
variable  v : set β

-- 1ª demostración
-- ===============

example : f '' (s ∪ f ⁻¹' v) ⊆ f '' s ∪ v :=
begin
  intros y hy,
  cases hy with x hx,
  cases hx with hx1 fxy,
  cases hx1 with xs xv,
  { left,
    use x,
    split,
    { exact xs, },
    { exact fxy, }},
  { right,
    rw ← fxy,
    exact xv, },
end

-- 2ª demostración
-- ===============

example : f '' (s ∪ f ⁻¹' v) ⊆ f '' s ∪ v :=
begin
  rintros y ⟨x, xs | xv, fxy⟩,
  { left,
    use [x, xs, fxy], },
  { right,
    rw ← fxy,
    exact xv, },
end

-- 3ª demostración
-- ===============
```

```
example : f '' (s ∪ f ⁻¹' v) ⊆ f '' s ∪ v :=
begin
  rintros y ⟨x, xs | xv, fxy⟩;
  finish,
end
```

Se puede interactuar con las pruebas anteriores en esta sesión con Lean.

3.21. Intersección con la imagen inversa

```
-- ---------------------------------------------------------------------
-- Demostrar que
--    s ∩ f ⁻¹' v ⊆ f ⁻¹' (f '' s ∩ v)
-- ----------------------------------------------------------------------

import data.set.basic

open set

variables {α : Type*} {β : Type*}
variable  f : α → β
variable  s : set α
variable  v : set β

-- 1ª demostración
-- ===============

example : s ∩ f ⁻¹' v ⊆ f ⁻¹' (f '' s ∩ v) :=
begin
  intros x hx,
  rw mem_preimage,
  split,
  { apply mem_image_of_mem,
    exact hx.1, },
  { rw ← mem_preimage,
    exact hx.2, },
end

-- 2ª demostración
-- ===============

example : s ∩ f ⁻¹' v ⊆ f ⁻¹' (f '' s ∩ v) :=
begin
```

```
  rintros x ⟨xs, xv⟩,
  split,
  { exact mem_image_of_mem f xs, },
  { exact xv, },
end

-- 3ª demostración
-- ===============

example : s ∩ f ⁻¹' v ⊆ f ⁻¹' (f '' s ∩ v) :=
begin
  rintros x ⟨xs, xv⟩,
  exact ⟨mem_image_of_mem f xs, xv⟩,
end

-- 4ª demostración
-- ===============

example : s ∩ f ⁻¹' v ⊆ f ⁻¹' (f '' s ∩ v) :=
begin
  rintros x ⟨xs, xv⟩,
  show f x ∈ f '' s ∩ v,
  split,
  { use [x, xs, rfl] },
  { exact xv },
end

-- 5ª demostración
-- ===============

example : s ∩ f ⁻¹' v ⊆ f ⁻¹' (f '' s ∩ v) :=
inter_preimage_subset s v f
```

Se puede interactuar con las pruebas anteriores en esta sesión con Lean.

3.22. Unión con la imagen inversa

```
-- ----------------------------------------------------------------------
-- Demostrar que
--    s ∪ f ⁻¹' v ⊆ f ⁻¹' (f '' s ∪ v)
-- -----------------------------------------------------------------------

import data.set.basic
```

```
open set

variables {α : Type*} {β : Type*}
variable  f : α → β
variable  s : set α
variable  v : set β

-- 1ª demostración
-- ===============

example : s ∪ f ⁻¹' v ⊆ f ⁻¹' (f '' s ∪ v) :=
begin
  intros x hx,
  rw mem_preimage,
  cases hx with xs xv,
  { apply mem_union_left,
    apply mem_image_of_mem,
    exact xs, },
  { apply mem_union_right,
    rw ← mem_preimage,
    exact xv, },
end

-- 2ª demostración
-- ===============

example : s ∪ f ⁻¹' v ⊆ f ⁻¹' (f '' s ∪ v) :=
begin
  intros x hx,
  cases hx with xs xv,
  { apply mem_union_left,
    apply mem_image_of_mem,
    exact xs, },
  { apply mem_union_right,
    exact xv, },
end

-- 3ª demostración
-- ===============

example : s ∪ f ⁻¹' v ⊆ f ⁻¹' (f '' s ∪ v) :=
begin
  rintros x (xs | xv),
  { left,
```

```
    exact mem_image_of_mem f xs, },
  { right,
    exact xv, },
end

-- 4ª demostración
-- ===============

example : s ∪ f ⁻¹' v ⊆ f ⁻¹' (f '' s ∪ v) :=
begin
  rintros x (xs | xv),
  { exact or.inl (mem_image_of_mem f xs), },
  { exact or.inr xv, },
end

-- 5ª demostración
-- ===============

example : s ∪ f ⁻¹' v ⊆ f ⁻¹' (f '' s ∪ v) :=
begin
  intros x h,
  exact or.elim h (λ xs, or.inl (mem_image_of_mem f xs)) or.inr,
end

-- 6ª demostración
-- ===============

example : s ∪ f ⁻¹' v ⊆ f ⁻¹' (f '' s ∪ v) :=
λ x h, or.elim h (λ xs, or.inl (mem_image_of_mem f xs)) or.inr

-- 7ª demostración
-- ===============

example : s ∪ f ⁻¹' v ⊆ f ⁻¹' (f '' s ∪ v) :=
begin
  rintros x (xs | xv),
  { show f x ∈ f '' s ∪ v,
    use [x, xs, rfl] },
  { show f x ∈ f '' s ∪ v,
    right,
    apply xv },
end

-- 8ª demostración
-- ===============
```

```
example : s ∪ f ⁻¹' v ⊆ f ⁻¹' (f '' s ∪ v) :=
union_preimage_subset s v f
```

Se puede interactuar con las pruebas anteriores en esta sesión con Lean.

3.23. Imagen de la unión general

```
-- ----------------------------------------------------------------------
-- Demostrar que
--    f '' (⋃ i, A i) = ⋃ i, f '' A i
-- ----------------------------------------------------------------------

import data.set.basic
import tactic

open set

variables {α : Type*} {β : Type*} {I : Type*}
variable  f : α → β
variables A : ℕ → set α

-- 1ª demostración
-- ===============

example : f '' (⋃ i, A i) = ⋃ i, f '' A i :=
begin
  ext y,
  split,
  { intro hy,
    rw mem_image at hy,
    cases hy with x hx,
    cases hx with xUA fxy,
    rw mem_Union at xUA,
    cases xUA with i xAi,
    rw mem_Union,
    use i,
    rw ← fxy,
    apply mem_image_of_mem,
    exact xAi, },
  { intro hy,
    rw mem_Union at hy,
    cases hy with i yAi,
```

```
    cases yAi with x hx,
    cases hx with xAi fxy,
    rw ← fxy,
    apply mem_image_of_mem,
    rw mem_Union,
    use i,
    exact xAi, },
end

-- 2ª demostración
-- ===============

example : f '' (⋃ i, A i) = ⋃ i, f '' A i :=
begin
  ext y,
  simp,
  split,
  { rintros ⟨x, ⟨i, xAi⟩, fxy⟩,
    use [i, x, xAi, fxy] },
  { rintros ⟨i, x, xAi, fxy⟩,
    exact ⟨x, ⟨i, xAi⟩, fxy⟩ },
end

-- 3ª demostración
-- ===============

example : f '' (⋃ i, A i) = ⋃ i, f '' A i :=
by tidy

-- 4ª demostración
-- ===============

example : f '' (⋃ i, A i) = ⋃ i, f '' A i :=
image_Union
```

Se puede interactuar con las pruebas anteriores en esta sesión con Lean.

3.24. Imagen de la intersección general

```
-- ----------------------------------------------------------------------
-- Demostrar que
--    f '' (⋂ i, A i) ⊆ ⋂ i, f '' A i
-- ----------------------------------------------------------------------
```

```
import data.set.basic
import tactic

open set

variables {α : Type*} {β : Type*} {I : Type*}
variable  f : α → β
variables A : ℕ → set α

-- 1ª demostración
-- ===============

example : f '' (⋂ i, A i) ⊆ ⋂ i, f '' A i :=
begin
  intros y h,
  apply mem_Inter_of_mem,
  intro i,
  cases h with x hx,
  cases hx with xIA fxy,
  rw ← fxy,
  apply mem_image_of_mem,
  exact mem_Inter.mp xIA i,
end

-- 2ª demostración
-- ===============

example : f '' (⋂ i, A i) ⊆ ⋂ i, f '' A i :=
begin
  intros y h,
  apply mem_Inter_of_mem,
  intro i,
  rcases h with ⟨x, xIA, rfl⟩,
  exact mem_image_of_mem f (mem_Inter.mp xIA i),
end

-- 3ª demostración
-- ===============

example : f '' (⋂ i, A i) ⊆ ⋂ i, f '' A i :=
begin
  intro y,
  simp,
  intros x xIA fxy i,
```

```
  use [x, xIA i, fxy],
end

-- 4ª demostración
-- ===============

example : f '' (⋂ i, A i) ⊆ ⋂ i, f '' A i :=
by tidy
```

Se puede interactuar con las pruebas anteriores en esta sesión con Lean.

3.25. Imagen de la intersección general mediante inyectiva

```
-- ----------------------------------------------------------------------
-- Demostrar que si f es inyectiva, entonces
--    (⋂ i, f '' A i) ⊆ f '' (⋂ i, A i)
-- -----------------------------------------------------------------------

import data.set.basic
import tactic

open set function

variables {α : Type*} {β : Type*} {I : Type*}
variable  f : α → β
variables A : I → set α

-- 1ª demostración
-- ===============

example
  (i : I)
  (injf : injective f)
  : (⋂ i, f '' A i) ⊆ f '' (⋂ i, A i) :=
begin
  intros y hy,
  rw mem_Inter at hy,
  rcases hy i with ⟨x, xAi, fxy⟩,
  use x,
  split,
  { apply mem_Inter_of_mem,
```

```
    intro j,
    rcases hy j with ⟨z, zAj, fzy⟩,
    convert zAj,
    apply injf,
    rw fxy,
    rw ← fzy, },
  { exact fxy, },
end

-- 2ª demostración
-- ===============

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, fxy⟩,
  use x,
  split,
  { intro j,
    rcases h j with ⟨z, zAi, fzy⟩,
    have : f x = f z, by rw [fxy, fzy],
    have : x = z, from injf this,
    rw this,
    exact zAi, },
  { exact fxy, },
end
```

Se puede interactuar con las pruebas anteriores en esta sesión con Lean.

3.26. Imagen inversa de la unión general

```
-- ---------------------------------------------------------------------
-- Demostrar que
--    f ⁻¹' (⋃ i, B i) = ⋃ i, f ⁻¹' (B i)
-- ----------------------------------------------------------------------

import data.set.basic
import tactic
```

```
open set

variables {α : Type*} {β : Type*} {I : Type*}
variable  f : α → β
variables B : I → set β

-- 1ª demostración
-- ===============

example : f ⁻¹' (⋃ i, B i) = ⋃ i, f ⁻¹' (B i) :=
begin
  ext x,
  split,
  { intro hx,
    rw mem_preimage at hx,
    rw mem_Union at hx,
    cases hx with i fxBi,
    rw mem_Union,
    use i,
    apply mem_preimage.mpr,
    exact fxBi, },
  { intro hx,
    rw mem_preimage,
    rw mem_Union,
    rw mem_Union at hx,
    cases hx with i xBi,
    use i,
    rw mem_preimage at xBi,
    exact xBi, },
end

-- 2ª demostración
-- ===============

example : f ⁻¹' (⋃ i, B i) = ⋃ i, f ⁻¹' (B i) :=
preimage_Union

-- 3ª demostración
-- ===============

example : f ⁻¹' (⋃ i, B i) = ⋃ i, f ⁻¹' (B i) :=
by  simp
```

Se puede interactuar con las pruebas anteriores en esta sesión con Lean.

3.27. Imagen inversa de la intersección general

```
-- ----------------------------------------------------------------------
-- Demostrar que
--    f ⁻¹' (⋂ i, B i) = ⋂ i, f ⁻¹' (B i)
-- ----------------------------------------------------------------------

import data.set.basic
import tactic

open set

variables {α : Type*} {β : Type*} {I : Type*}
variable  f : α → β
variables B : I → set β

-- 1ª demostración
-- ===============

example : f ⁻¹' (⋂ i, B i) = ⋂ i, f ⁻¹' (B i) :=
begin
  ext x,
  split,
  { intro hx,
    apply mem_Inter_of_mem,
    intro i,
    rw mem_preimage,
    rw mem_preimage at hx,
    rw mem_Inter at hx,
    exact hx i, },
  { intro hx,
    rw mem_preimage,
    rw mem_Inter,
    intro i,
    rw ← mem_preimage,
    rw mem_Inter at hx,
    exact hx i, },
end

-- 2ª demostración
-- ===============

example : f ⁻¹' (⋂ i, B i) = ⋂ i, f ⁻¹' (B i) :=
```

```
begin
  ext x,
  calc  (x ∈ f ⁻¹' ⋂ (i : I), B i)
      ↔ f x ∈ ⋂ (i : I), B i        : mem_preimage
  ... ↔ (∀ i : I, f x ∈ B i)        : mem_Inter
  ... ↔ (∀ i : I, x ∈ f ⁻¹' B i)    : iff_of_eq rfl
  ... ↔ x ∈ ⋂ (i : I), f ⁻¹' B i    : mem_Inter.symm,
end

-- 3ª demostración
-- ===============

example : f ⁻¹' (⋂ i, B i) = ⋂ i, f ⁻¹' (B i) :=
begin
  ext x,
  simp,
end

-- 4ª demostración
-- ===============

example : f ⁻¹' (⋂ i, B i) = ⋂ i, f ⁻¹' (B i) :=
by { ext, simp }
```

Se puede interactuar con las pruebas anteriores en esta sesión con Lean.

3.28. Teorema de Cantor

```
-- ---------------------------------------------------------------------
-- Demostrar el teorema de Cantor:
--    ∀ f : α → set α, ¬ surjective f
-- ----------------------------------------------------------------------

import data.set.basic
open function

variables {α : Type}

-- 1ª demostración
-- ===============

example : ∀ f : α → set α, ¬ surjective f :=
```

```
begin
  intros f surjf,
  let S := {i | i ∉ f i},
  unfold surjective at surjf,
  cases surjf S with j fjS,
  by_cases j ∈ S,
  { apply absurd _ h,
    rw fjS,
    exact h, },
  { apply h,
    rw ← fjS at h,
    exact h, },
end

-- 2ª demostración
-- ===============

example : ∀ f : α → set α, ¬ surjective f :=
begin
  intros f surjf,
  let S := {i | i ∉ f i},
  cases surjf S with j fjS,
  by_cases j ∈ S,
  { apply absurd _ h,
    rwa fjS, },
  { apply h,
    rwa ← fjS at h, },
end

-- 3ª demostración
-- ===============

example : ∀ f : α → set α, ¬ surjective f :=
begin
  intros f surjf,
  let S := {i | i ∉ f i},
  cases surjf S with j fjS,
  have h : (j ∈ S) = (j ∉ S), from
    calc  (j ∈ S)
        = (j ∉ f j) : set.mem_set_of_eq
    ... = (j ∉ S)   : congr_arg not (congr_arg (has_mem.mem j) fjS),
  exact false_of_a_eq_not_a h,
end
```

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

example : ∀ f : α → set α, ¬ surjective f :=
cantor_surjective
```

Se puede interactuar con las pruebas anteriores en esta sesión con Lean.

3.29. En los monoides, los inversos a la izquierda y a la derecha son iguales

```
-- En_los_monoides_los_inversos_a_la_izquierda_y_a_la_derecha_son_iguales.lean
-- En los monoides, los inversos a la izquierda y a la derecha son iguales.
-- José A. Alonso Jiménez
-- Sevilla, 29 de junio de 2021
-- ---------------------------------------------------------------------

-- ---------------------------------------------------------------------
-- Un [monoide](https://en.wikipedia.org/wiki/Monoid) es un conjunto
-- junto con una operación binaria que es asociativa y tiene elemento
-- neutro.
--
-- En Lean, está definida la clase de los monoides (como `monoid`) y sus
-- propiedades características son
--    mul_assoc : (a * b) * c = a * (b * c)
--    one_mul :   1 * a = a
--    mul_one :   a * 1 = a
--
-- Demostrar que si M es un monide, a ∈ M, b es un inverso de a por la
-- izquierda y c es un inverso de a por la derecha, entonce b = c.
-- ---------------------------------------------------------------------

import algebra.group.defs

variables {M : Type} [monoid M]
variables {a b c : M}

-- 1ª demostración
-- ===============

example
  (hba : b * a = 1)
```

```
  (hac : a * c = 1)
  : b = c :=
begin
 rw ←one_mul c,
 rw ←hba,
 rw mul_assoc,
 rw hac,
 rw mul_one b,
end

-- 2ª demostración
-- ===============

example
  (hba : b * a = 1)
  (hac : a * c = 1)
  : b = c :=
by rw [←one_mul c, ←hba, mul_assoc, hac, mul_one b]

-- 3ª demostración
-- ===============

example
  (hba : b * a = 1)
  (hac : a * c = 1)
  : b = c :=
calc b   = b * 1       : (mul_one b).symm
     ... = b * (a * c) : congr_arg (λ x, b * x) hac.symm
     ... = (b * a) * c : (mul_assoc b a c).symm
     ... = 1 * c       : congr_arg (λ x, x * c) hba
     ... = c           : one_mul c

-- 4ª demostración
-- ===============

example
  (hba : b * a = 1)
  (hac : a * c = 1)
  : b = c :=
calc b   = b * 1       : by finish
     ... = b * (a * c) : by finish
     ... = (b * a) * c : (mul_assoc b a c).symm
     ... = 1 * c       : by finish
     ... = c           : by finish
```

```
-- 5ª demostración
-- ===============

example
  (hba : b * a = 1)
  (hac : a * c = 1)
  : b = c :=
left_inv_eq_right_inv hba hac
```

”7-”Se puede interactuar con las pruebas anteriores en esta sesión con Lean.

3.30. Producto_de_potencias_de_la_misma_base_en_r

```
-- ---------------------------------------------------------------------
-- En los [monoides](https://en.wikipedia.org/wiki/Monoid) se define la
-- potencia con exponentes naturales. En Lean la potencia x^n se
-- se caracteriza por los siguientes lemas:
--    pow_zero : x^0 = 1
--    pow_succ : x^(succ n) = x * x^n
--
-- Demostrar que
--    x^(m + n) = x^m * x^n
-- ---------------------------------------------------------------------

import algebra.group_power.basic
open monoid nat

variables {M : Type} [monoid M]
variable  x : M
variables (m n : ℕ)

-- Para que no use la notación con puntos
set_option pp.structure_projections false

-- 1ª demostración
-- ===============

example :
  x^(m + n) = x^m * x^n :=
begin
  induction m with m HI,
  { calc x^(0 + n)
         = x^n              : congr_arg ((^) x) (nat.zero_add n)
```

```
    ... = 1 * x^n          : (monoid.one_mul (x^n)).symm
    ... = x^0 * x^n        : congr_arg (* (x^n)) (pow_zero x).symm, },
  { calc x^(succ m + n)
        = x^succ (m + n)   : congr_arg ((^) x) (succ_add m n)
    ... = x * x^(m + n)    : pow_succ x (m + n)
    ... = x * (x^m * x^n) : congr_arg ((*) x) HI
    ... = (x * x^m) * x^n : (monoid.mul_assoc x (x^m) (x^n)).symm
    ... = x^succ m * x^n   : congr_arg (* x^n) (pow_succ x m).symm, },
end

-- 2ª demostración
-- ===============

example :
  x^(m + n) = x^m * x^n :=
begin
  induction m with m HI,
  { calc x^(0 + n)
        = x^n              : by simp only [nat.zero_add]
    ... = 1 * x^n          : by simp only [monoid.one_mul]
    ... = x^0 * x^n        : by simp [pow_zero] },
  { calc x^(succ m + n)
        = x^succ (m + n)   : by simp only [succ_add]
    ... = x * x^(m + n)    : by simp only [pow_succ]
    ... = x * (x^m * x^n) : by simp only [HI]
    ... = (x * x^m) * x^n : (monoid.mul_assoc x (x^m) (x^n)).symm
    ... = x^succ m * x^n   : by simp only [pow_succ], },
end

-- 3ª demostración
-- ===============

example :
  x^(m + n) = x^m * x^n :=
begin
  induction m with m HI,
  { calc x^(0 + n)
        = x^n              : by simp [nat.zero_add]
    ... = 1 * x^n          : by simp
    ... = x^0 * x^n        : by simp, },
  { calc x^(succ m + n)
        = x^succ (m + n)   : by simp [succ_add]
```

```
     ... = x * x^(m + n)    : by simp [pow_succ]
     ... = x * (x^m * x^n) : by simp [HI]
     ... = (x * x^m) * x^n : (monoid.mul_assoc x (x^m) (x^n)).symm
     ... = x^succ m * x^n  : by simp [pow_succ], },
end

-- 4ª demostración
-- ===============

example :
  x^(m + n) = x^m * x^n :=
begin
  induction m with m HI,
  { show x^(0 + n) = x^0 * x^n,
      by simp [nat.zero_add] },
  { show x^(succ m + n) = x^succ m * x^n,
      by finish [succ_add,
                 HI,
                 monoid.mul_assoc,
                 pow_succ], },
end

-- 5ª demostración
-- ===============

example :
  x^(m + n) = x^m * x^n :=
pow_add x m n
```

Se puede interactuar con las pruebas anteriores en esta sesión con Lean.

Capítulo 4

Ejercicios de julio de 2021

4.1. Equivalencia de inversos iguales al neutro

```
-- ----------------------------------------------------------------------
-- Sea M un monoide y a, b ∈ M tales que a * b = 1. Demostrar que a = 1
-- si y sólo si b = 1.
-- ----------------------------------------------------------------------

import algebra.group.basic

variables {M : Type} [monoid M]
variables {a b : M}

-- 1ª demostración
-- ===============

example
  (h : a * b = 1)
  : a = 1 ↔ b = 1 :=
begin
  split,
  { intro a1,
    rw a1 at h,
    rw one_mul at h,
    exact h, },
  { intro b1,
    rw b1 at h,
    rw mul_one at h,
    exact h, },
end
```

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

example
  (h : a * b = 1)
  : a = 1 ↔ b = 1 :=
begin
  split,
  { intro a1,
    calc b = 1 * b : (one_mul b).symm
       ... = a * b : congr_arg (* b) a1.symm
       ... = 1     : h, },
  { intro b1,
    calc a = a * 1 : (mul_one a).symm
       ... = a * b : congr_arg ((*) a) b1.symm
       ... = 1     : h, },
end

-- 3ª demostración
-- ===============

example
  (h : a * b = 1)
  : a = 1 ↔ b = 1 :=
begin
  split,
  { rintro rfl,
    simpa using h, },
  { rintro rfl,
    simpa using h, },
end

-- 4ª demostración
-- ===============

example
  (h : a * b = 1)
  : a = 1 ↔ b = 1 :=
by split ; { rintro rfl, simpa using h }

-- 5ª demostración
-- ===============

example
  (h : a * b = 1)
```

```
  : a = 1 ↔ b = 1 :=
by split ; finish

-- 6ª demostración
-- ===============

example
  (h : a * b = 1)
  : a = 1 ↔ b = 1 :=
by finish [iff_def]

-- 7ª demostración
-- ===============

example
  (h : a * b = 1)
  : a = 1 ↔ b = 1 :=
eq_one_iff_eq_one_of_mul_eq_one h
```

Se puede interactuar con las pruebas anteriores en esta sesión con Lean.

4.2. Unicidad de inversos en monoides

```
-- ---------------------------------------------------------------------
-- Demostrar que en los monoides conmutativos, si un elemento tiene un
-- inverso por la derecha, dicho inverso es único.
-- ---------------------------------------------------------------------

import algebra.group.basic
import tactic

variables {M : Type} [comm_monoid M]
variables {x y z : M}

-- 1ª demostración
-- ===============

example
  (hy : x * y = 1)
  (hz : x * z = 1)
  : y = z :=
calc y = 1 * y       : (one_mul y).symm
   ... = (x * z) * y : congr_arg (* y) hz.symm
```

```
   ... = (z * x) * y : congr_arg (* y) (mul_comm x z)
   ... = z * (x * y) : mul_assoc z x y
   ... = z * 1       : congr_arg ((*) z) hy
   ... = z           : mul_one z

-- 2ª demostración
-- ===============

example
  (hy : x * y = 1)
  (hz : x * z = 1)
  : y = z :=
calc y = 1 * y       : by simp only [one_mul]
   ... = (x * z) * y : by simp only [hz]
   ... = (z * x) * y : by simp only [mul_comm]
   ... = z * (x * y) : by simp only [mul_assoc]
   ... = z * 1       : by simp only [hy]
   ... = z           : by simp only [mul_one]

-- 3ª demostración
-- ===============

example
  (hy : x * y = 1)
  (hz : x * z = 1)
  : y = z :=
calc y = 1 * y       : by simp
   ... = (x * z) * y : by simp [hz]
   ... = (z * x) * y : by simp [mul_comm]
   ... = z * (x * y) : by simp [mul_assoc]
   ... = z * 1       : by simp [hy]
   ... = z           : by simp

-- 4ª demostración
-- ===============

example
  (hy : x * y = 1)
  (hz : x * z = 1)
  : y = z :=
begin
  apply left_inv_eq_right_inv _ hz,
  rw mul_comm,
  exact hy,
end
```

```
-- 5ª demostración
-- ===============

example
  (hy : x * y = 1)
  (hz : x * z = 1)
  : y = z :=
left_inv_eq_right_inv (trans (mul_comm _ _) hy) hz

-- 6ª demostración
-- ===============

example
  (hy : x * y = 1)
  (hz : x * z = 1)
  : y = z :=
inv_unique hy hz
```

Se puede interactuar con las pruebas anteriores en esta sesión con Lean.

4.3. Caracterización de producto igual al primer factor

```
-- ---------------------------------------------------------------------
-- Un monoide cancelativo por la izquierda es un monoide
-- https://bit.ly/3h4notA M que cumple la propiedad cancelativa por la
-- izquierda; es decir, para todo a, b ∈ M
--    a * b = a * c ↔ b = c.
--
-- En Lean la clase de los monoides cancelativos por la izquierda es
-- left_cancel_monoid y la propiedad cancelativa por la izquierda es
--    mul_left_cancel_iff : a * b = a * c ↔ b = c
--
-- Demostrar que si M es un monoide cancelativo por la izquierda y
-- a, b ∈ M, entonces
--    a * b = a ↔ b = 1
-- ---------------------------------------------------------------------

import algebra.group.basic

universe  u
variables {M : Type u} [left_cancel_monoid M]
```

```
variables {a b : M}

-- ?ª demostración
-- ===============

example : a * b = a ↔ b = 1 :=
begin
  split,
  { intro h,
    rw ← @mul_left_cancel_iff _ _ a b 1,
    rw mul_one,
    exact h, },
  { intro h,
    rw h,
    exact mul_one a, },
end

-- ?ª demostración
-- ===============

example : a * b = a ↔ b = 1 :=
calc a * b = a ↔ a * b = a * 1 : by rw mul_one
          ... ↔ b = 1         : mul_left_cancel_iff

-- ?ª demostración
-- ===============

example : a * b = a ↔ b = 1 :=
mul_right_eq_self

-- ?ª demostración
-- ===============

example : a * b = a ↔ b = 1 :=
by finish
```

Se puede interactuar con las pruebas anteriores en esta sesión con Lean.

4.4. Unicidad del elemento neutro en los grupos

```
-- ---------------------------------------------------------------------
-- Demostrar que un grupo sólo posee un elemento neutro.
-- ---------------------------------------------------------------------

import algebra.group.basic

universe  u
variables {G : Type u} [group G]

-- 1ª demostración
-- ===============

example
  (e : G)
  (h : ∀ x, x * e = x)
  : e = 1 :=
calc e = 1 * e : (one_mul e).symm
   ... = 1     : h 1

-- 2ª demostración
-- ===============

example
  (e : G)
  (h : ∀ x, x * e = x)
  : e = 1 :=
self_eq_mul_left.mp (congr_arg _ (congr_arg _ (eq.symm (h e))))

-- 3ª demostración
-- ===============

example
  (e : G)
  (h : ∀ x, x * e = x)
  : e = 1 :=
by finish

-- Referencia
-- ==========

-- Propiedad 3.17 del libro "Abstract algebra: Theory and applications"
```

```
-- de Thomas W. Judson.
-- http://abstract.ups.edu/download/aata-20200730.pdf#page=49
```

Se puede interactuar con las pruebas anteriores en esta sesión con Lean.

4.5. Unicidad de los inversos en los grupos

```
-- ---------------------------------------------------------------------
-- Demostrar que si a es un elemento de un grupo G, entonces a tiene un
-- único inverso; es decir, si b es un elemento de G tal que a * b = 1,
-- entonces a⁻¹ = b.
-- ---------------------------------------------------------------------

import algebra.group.basic

universe  u
variables {G : Type u} [group G]
variables {a b : G}

-- 1ª demostración
-- ===============

example
  (h : a * b = 1)
  : a⁻¹ = b :=
calc a⁻¹ = a⁻¹ * 1       : (mul_one a⁻¹).symm
     ... = a⁻¹ * (a * b) : congr_arg ((*) a⁻¹) h.symm
     ... = (a⁻¹ * a) * b : (mul_assoc a⁻¹ a b).symm
     ... = 1 * b         : congr_arg (* b) (inv_mul_self a)
     ... = b             : one_mul b

-- 2ª demostración
-- ===============

example
  (h : a * b = 1)
  : a⁻¹ = b :=
calc a⁻¹ = a⁻¹ * 1       : by simp only [mul_one]
     ... = a⁻¹ * (a * b) : by simp only [h]
     ... = (a⁻¹ * a) * b : by simp only [mul_assoc]
     ... = 1 * b         : by simp only [inv_mul_self]
     ... = b             : by simp only [one_mul]
```

```
-- 3ª demostración
-- ===============

example
  (h : a * b = 1)
  : a⁻¹ = b :=
calc a⁻¹ = a⁻¹ * 1       : by simp
     ... = a⁻¹ * (a * b) : by simp [h]
     ... = (a⁻¹ * a) * b : by simp
     ... = 1 * b         : by simp
     ... = b             : by simp

-- 4ª demostración
-- ===============

example
  (h : a * b = 1)
  : a⁻¹ = b :=
calc a⁻¹ = a⁻¹ * (a * b) : by simp [h]
     ... = b             : by simp

-- 5ª demostración
-- ===============

example
  (h : b * a = 1)
  : b = a⁻¹ :=
eq_inv_of_mul_eq_one h

-- Referencia
-- ==========

-- Propiedad 3.18 del libro "Abstract algebra: Theory and applications"
-- de Thomas W. Judson.
-- http://abstract.ups.edu/download/aata-20200730.pdf#page=49
```

Se puede interactuar con las pruebas anteriores en esta sesión con Lean.

4.6. Inverso del producto

```
-- ---------------------------------------------------------------------
-- Sea G un grupo y a, b ∈ G. Entonces,
--    (a * b)⁻¹ = b⁻¹ * a⁻¹
```

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

import algebra.group.basic

universe  u
variables {G : Type u} [group G]
variables {a b : G}

-- 1ª demostración
-- ===============

example : (a * b)⁻¹ = b⁻¹ * a⁻¹ :=
begin
  apply inv_eq_of_mul_eq_one,
  calc a * b * (b⁻¹ * a⁻¹)
       = ((a * b) * b⁻¹) * a⁻¹ : (mul_assoc _ _ _).symm
   ... = (a * (b * b⁻¹)) * a⁻¹ : congr_arg (* a⁻¹) (mul_assoc a _ _)
   ... = (a * 1) * a⁻¹         : congr_arg2 _ (congr_arg _ (mul_inv_self b)) rfl
   ... = a * a⁻¹               : congr_arg (* a⁻¹) (mul_one a)
   ... = 1                     : mul_inv_self a
end

-- 2ª demostración
-- ===============

example : (a * b)⁻¹ = b⁻¹ * a⁻¹ :=
begin
  apply inv_eq_of_mul_eq_one,
  calc a * b * (b⁻¹ * a⁻¹)
       = ((a * b) * b⁻¹) * a⁻¹ : by simp only [mul_assoc]
   ... = (a * (b * b⁻¹)) * a⁻¹ : by simp only [mul_assoc]
   ... = (a * 1) * a⁻¹         : by simp only [mul_inv_self]
   ... = a * a⁻¹               : by simp only [mul_one]
   ... = 1                     : by simp only [mul_inv_self]
end

-- 3ª demostración
-- ===============

example : (a * b)⁻¹ = b⁻¹ * a⁻¹ :=
begin
  apply inv_eq_of_mul_eq_one,
  calc a * b * (b⁻¹ * a⁻¹)
       = ((a * b) * b⁻¹) * a⁻¹ : by simp [mul_assoc]
   ... = (a * (b * b⁻¹)) * a⁻¹ : by simp
```

```
   ... = (a * 1) * a⁻¹         : by simp
   ... = a * a⁻¹               : by simp
   ... = 1                     : by simp,
end

-- 4ª demostración
-- ===============

example : (a * b)⁻¹ = b⁻¹ * a⁻¹ :=
mul_inv_rev a b

-- 5ª demostración
-- ===============

example : (a * b)⁻¹ = b⁻¹ * a⁻¹ :=
by simp

-- Referencia
-- ==========

-- Propiedad 3.19 del libro "Abstract algebra: Theory and applications"
-- de Thomas W. Judson.
-- http://abstract.ups.edu/download/aata-20200730.pdf#page=49
```

Se puede interactuar con las pruebas anteriores en esta sesión con Lean.

4.7. Inverso del inverso en grupos

```
-- ----------------------------------------------------------------------
-- Sea G un grupo y a ∈ G. Demostrar que
--    (a⁻¹)⁻¹ = a
-- ----------------------------------------------------------------------

import algebra.group.basic

universe  u
variables {G : Type u} [group G]
variables {a b : G}

-- 1ª demostración
-- ===============

example : (a⁻¹)⁻¹ = a :=
```

```
calc (a⁻¹)⁻¹
      = (a⁻¹)⁻¹ * 1           : (mul_one (a⁻¹)⁻¹).symm
  ... = (a⁻¹)⁻¹ * (a⁻¹ * a) : congr_arg ((*) (a⁻¹)⁻¹) (inv_mul_self a).symm
  ... = ((a⁻¹)⁻¹ * a⁻¹) * a : (mul_assoc _ _ _).symm
  ... = 1 * a                 : congr_arg (* a) (inv_mul_self a⁻¹)
  ... = a                     : one_mul a

-- 2ª demostración
-- ===============

example : (a⁻¹)⁻¹ = a :=
calc (a⁻¹)⁻¹
      = (a⁻¹)⁻¹ * 1           : by simp only [mul_one]
  ... = (a⁻¹)⁻¹ * (a⁻¹ * a) : by simp only [inv_mul_self]
  ... = ((a⁻¹)⁻¹ * a⁻¹) * a : by simp only [mul_assoc]
  ... = 1 * a                 : by simp only [inv_mul_self]
  ... = a                     : by simp only [one_mul]

-- 3ª demostración
-- ===============

example : (a⁻¹)⁻¹ = a :=
calc (a⁻¹)⁻¹
      = (a⁻¹)⁻¹ * 1           : by simp
  ... = (a⁻¹)⁻¹ * (a⁻¹ * a) : by simp
  ... = ((a⁻¹)⁻¹ * a⁻¹) * a : by simp
  ... = 1 * a                 : by simp
  ... = a                     : by simp

-- 4ª demostración
-- ===============

example : (a⁻¹)⁻¹ = a :=
begin
  apply inv_eq_of_mul_eq_one,
  exact mul_left_inv a,
end

-- 5ª demostración
-- ===============

example : (a⁻¹)⁻¹ = a :=
inv_eq_of_mul_eq_one (mul_left_inv a)

-- 6ª demostración
```

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

example : (a⁻¹)⁻¹ = a:=
inv_inv a

-- 7ª demostración
-- ===============

example : (a⁻¹)⁻¹ = a:=
by simp


-- Referencia
-- ==========

-- Propiedad 3.20 del libro "Abstract algebra: Theory and applications"
-- de Thomas W. Judson.
-- http://abstract.ups.edu/download/aata-20200730.pdf#page=49
```

Se puede interactuar con las pruebas anteriores en esta sesión con Lean.

4.8. Propiedad cancelativa en grupos

```
-- ----------------------------------------------------------------------
-- Sea G un grupo y a,b,c ∈ G. Demostrar que si a * b = a* c, entonces
-- b = c.
-- ----------------------------------------------------------------------

import algebra.group.basic

universe  u
variables {G : Type u} [group G]
variables {a b c : G}

-- 1ª demostración
-- ===============

example
  (h: a * b = a  * c)
  : b = c :=
calc b = 1 * b          : (one_mul b).symm
   ... = (a⁻¹ * a) * b : congr_arg (* b) (inv_mul_self a).symm
   ... = a⁻¹ * (a * b) : mul_assoc a⁻¹ a b
```

```
    ... = a⁻¹ * (a * c) : congr_arg ((*) a⁻¹) h
    ... = (a⁻¹ * a) * c : (mul_assoc a⁻¹ a c).symm
    ... = 1 * c         : congr_arg (* c) (inv_mul_self a)
    ... = c             : one_mul c

-- 2ª demostración
-- ===============

example
  (h: a * b = a  * c)
  : b = c :=
calc b = 1 * b         : by rw one_mul
    ... = (a⁻¹ * a) * b : by rw inv_mul_self
    ... = a⁻¹ * (a * b) : by rw mul_assoc
    ... = a⁻¹ * (a * c) : by rw h
    ... = (a⁻¹ * a) * c : by rw mul_assoc
    ... = 1 * c         : by rw inv_mul_self
    ... = c             : by rw one_mul

-- 3ª demostración
-- ===============

example
  (h: a * b = a  * c)
  : b = c :=
calc b = 1 * b         : by simp
    ... = (a⁻¹ * a) * b : by simp
    ... = a⁻¹ * (a * b) : by simp
    ... = a⁻¹ * (a * c) : by simp [h]
    ... = (a⁻¹ * a) * c : by simp
    ... = 1 * c         : by simp
    ... = c             : by simp

-- 4ª demostración
-- ===============

example
  (h: a * b = a  * c)
  : b = c :=
calc b = a⁻¹ * (a * b) : by simp
    ... = a⁻¹ * (a * c) : by simp [h]
    ... = c             : by simp

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

```
example
  (h: a * b = a  * c)
  : b = c :=
begin
  have h1 : a⁻¹ * (a * b) = a⁻¹ * (a  * c),
    { by finish [h] },
  have h2 : (a⁻¹ * a) * b = (a⁻¹ * a)  * c,
    { by finish },
  have h3 : 1 * b = 1  * c,
    { by finish },
  have h3 : b = c,
    { by finish },
  exact h3,
end

-- 4ª demostración
-- ===============

example
  (h: a * b = a  * c)
  : b = c :=
begin
  have : a⁻¹ * (a * b) = a⁻¹ * (a  * c),
    { by finish [h] },
  have h2 : (a⁻¹ * a) * b = (a⁻¹ * a)  * c,
    { by finish },
  have h3 : 1 * b = 1  * c,
    { by finish },
  have h3 : b = c,
    { by finish },
  exact h3,
end

-- 4ª demostración
-- ===============

example
  (h: a * b = a  * c)
  : b = c :=
begin
  have h1 : a⁻¹ * (a * b) = a⁻¹ * (a  * c),
    { congr, exact h, },
  have h2 : (a⁻¹ * a) * b = (a⁻¹ * a)  * c,
    { simp only [h1, mul_assoc], },
```

```
  have h3 : 1 * b = 1  * c,
    { simp only [h2, (inv_mul_self a).symm], },
  rw one_mul at h3,
  rw one_mul at h3,
  exact h3,
end

-- 5ª demostración
-- ===============

example
  (h: a * b = a  * c)
  : b = c :=
mul_left_cancel h

-- 6ª demostración
-- ===============

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

-- Referencias
-- ===========

-- Propiedad 3.22 del libro "Abstract algebra: Theory and applications"
-- de Thomas W. Judson.
-- http://abstract.ups.edu/download/aata-20200730.pdf
```

Se puede interactuar con las pruebas anteriores en esta sesión con Lean.

4.9. Potencias de potencias en monoides

```
-- ---------------------------------------------------------------------
-- En los [monoides](https://en.wikipedia.org/wiki/Monoid) se define la
-- potencia con exponentes naturales. En Lean la potencia x^n se
-- se caracteriza por los siguientes lemas:
--    pow_zero : x^0 = 1
--    pow_succ' : x^(succ n) = x * x^n
--
-- Demostrar que si M, a ∈ M y m, n ∈ ℕ, entonces
--    a^(m * n) = (a^m)^n
```

```
--
-- Indicación: Se puede usar el lema
--    pow_add : a^(m + n) = a^m * a^n
-- ----------------------------------------------------------------------

import algebra.group_power.basic
open monoid nat

variables {M : Type} [monoid M]
variable  a : M
variables (m n : ℕ)

-- Para que no use la notación con puntos
set_option pp.structure_projections false

-- 1ª demostración
-- ===============

example : a^(m * n) = (a^m)^n :=
begin
  induction n with n HI,
  { calc a^(m * 0)
         = a^0              : congr_arg ((^) a) (nat.mul_zero m)
     ... = 1                : pow_zero a
     ... = (a^m)^0          : (pow_zero (a^m)).symm },
  { calc a^(m * succ n)
         = a^(m * n + m)    : congr_arg ((^) a) (nat.mul_succ m n)
     ... = a^(m * n) * a^m : pow_add a (m * n) m
     ... = (a^m)^n * a^m    : congr_arg (* a^m) HI
     ... = (a^m)^(succ n)   : (pow_succ' (a^m) n).symm },
end

-- 2ª demostración
-- ===============

example : a^(m * n) = (a^m)^n :=
begin
  induction n with n HI,
  { calc a^(m * 0)
         = a^0              : by simp only [nat.mul_zero]
     ... = 1                : by simp only [pow_zero]
     ... = (a^m)^0          : by simp only [pow_zero] },
  { calc a^(m * succ n)
         = a^(m * n + m)    : by simp only [nat.mul_succ]
```

```
    ... = a^(m * n) * a^m : by simp only [pow_add]
    ... = (a^m)^n * a^m   : by simp only [HI]
    ... = (a^m)^succ n    : by simp only [pow_succ'] },
end

-- 3ª demostración
-- ===============

example : a^(m * n) = (a^m)^n :=
begin
  induction n with n HI,
  { calc a^(m * 0)
        = a^0             : by simp [nat.mul_zero]
    ... = 1               : by simp
    ... = (a^m)^0         : by simp },
  { calc a^(m * succ n)
        = a^(m * n + m)   : by simp [nat.mul_succ]
    ... = a^(m * n) * a^m : by simp [pow_add]
    ... = (a^m)^n * a^m   : by simp [HI]
    ... = (a^m)^succ n    : by simp [pow_succ'] },
end

-- 4ª demostración
-- ===============

example : a^(m * n) = (a^m)^n :=
begin
  induction n with n HI,
  { by simp [nat.mul_zero] },
  { by simp [nat.mul_succ,
             pow_add,
             HI,
             pow_succ'] },
end

-- 5ª demostración
-- ===============

example : a^(m * n) = (a^m)^n :=
begin
  induction n with n HI,
  { rw nat.mul_zero,
    rw pow_zero,
```

```
    rw pow_zero, },
  { rw nat.mul_succ,
    rw pow_add,
    rw HI,
    rw pow_succ', }
end

-- 6ª demostración
-- ===============

example : a^(m * n) = (a^m)^n :=
begin
  induction n with n HI,
  { rw [nat.mul_zero, pow_zero, pow_zero] },
  { rw [nat.mul_succ, pow_add, HI, pow_succ'] }
end

-- 7ª demostración
-- ===============

example : a^(m * n) = (a^m)^n :=
pow_mul a m n
```

Se puede interactuar con las pruebas anteriores en esta sesión con Lean.

4.10. Los monoides booleanos son conmutativos

```
begin
  intros a b,
  calc a * b
      = (a * b) * 1
        : (mul_one (a * b)).symm
   ... = (a * b) * (a * a)
        : congr_arg ((*) (a*b)) (h a).symm
   ... = ((a * b) * a) * a
        : (mul_assoc (a*b) a a).symm
   ... = (a * (b * a)) * a
        : congr_arg (* a) (mul_assoc a b a)
   ... = (1 * (a * (b * a))) * a
        : congr_arg (* a) (one_mul (a*(b*a))).symm
   ... = ((b * b) * (a * (b * a))) * a
```

```
          : congr_arg (* a) (congr_arg (* (a*(b*a))) (h b).symm)
   ... = (b * (b * (a * (b * a)))) * a
          : congr_arg (* a) (mul_assoc b b (a*(b*a)))
   ... = (b * ((b * a) * (b * a))) * a
          : congr_arg (* a) (congr_arg ((*) b) (mul_assoc b a (b*a)).symm)
   ... = (b * 1) * a
          : congr_arg (* a) (congr_arg ((*) b) (h (b*a)))
   ... = b * a
          : congr_arg (* a) (mul_one b),
end
```

Se puede interactuar con las pruebas anteriores en esta sesión con Lean.

4.11. Límite de sucesiones constantes

```
-- ----------------------------------------------------------------------
-- En Lean, una sucesión u₀, u₁, u₂, ... se puede representar mediante
-- una función (u : ℕ → ℝ) de forma que u(n) es uₙ.
--
-- Se define que a es el límite de la sucesión u, por
--    def limite : (ℕ → ℝ) → ℝ → Prop :=
--    λ u a, ∀ ε > 0, ∃ N, ∀ n ≥ N, |u n - a| < ε
-- donde se usa la notación |x| para el valor absoluto de x
--    notation `|`x`|` := abs x
--
-- Demostrar que el límite de la sucesión constante c es c.
-- ----------------------------------------------------------------------

import data.real.basic

variable (u : ℕ → ℝ)
variable (c : ℝ)

notation `|`x`|` := abs x

def limite : (ℕ → ℝ) → ℝ → Prop :=
λ u a, ∀ ε > 0, ∃ N, ∀ n ≥ N, |u n - a| < ε

-- 1ª demostración
-- ===============

example :
  limite (λ n, c) c :=
```

```
begin
  unfold limite,
  intros ε hε,
  use 0,
  intros n hn,
  dsimp,
  simp,
  exact hε,
end

-- 2ª demostración
-- ===============

example :
  limite (λ n, c) c :=
begin
  intros ε hε,
  use 0,
  rintro n -,
  norm_num,
  assumption,
end

-- 3ª demostración
-- ===============

example :
  limite (λ n, c) c :=
begin
  intros ε hε,
  use 0,
  intros n hn,
  calc |(λ n, c) n - c|
         = |c - c|   : rfl
     ... = 0         : by simp
     ... < ε         : hε
end

-- 4ª demostración
-- ===============

example :
  limite (λ n, c) c :=
begin
  intros ε hε,
```

```
  by finish,
end

-- 5ª demostración
-- ===============

example :
  limite (λ n, c) c :=
λ ε hε, by finish

-- 6ª demostración
-- ===============

example :
  limite (λ n, c) c :=
assume ε,
assume hε : ε > 0,
exists.intro 0
  ( assume n,
    assume hn : n ≥ 0,
    show |(λ n, c) n - c| < ε, from
      calc |(λ n, c) n - c|
           = |c - c|   : rfl
       ... = 0         : by simp
       ... < ε         : hε)
```

Se puede interactuar con las pruebas anteriores en esta sesión con Lean.

4.12. Unicidad del límite de las sucesiones convergentes

```
-- ----------------------------------------------------------------------
-- En Lean, una sucesión u₀, u₁, u₂, ... se puede representar mediante
-- una función (u : ℕ → ℝ) de forma que u(n) es uₙ.
--
-- Se define que a es el límite de la sucesión u, por
--    def limite : (ℕ → ℝ) → ℝ → Prop :=
--    λ u a, ∀ ε > 0, ∃ N, ∀ n ≥ N, |u n - a| < ε
-- donde se usa la notación |x| para el valor absoluto de x
--    notation `|`x`|` := abs x
--
-- Demostrar que cada sucesión tiene como máximo un límite.
```

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

import data.real.basic

variables {u : ℕ → ℝ}
variables {a b : ℝ}

notation `|`x`|` := abs x

def limite : (ℕ → ℝ) → ℝ → Prop :=
λ u c, ∀ ε > 0, ∃ N, ∀ n ≥ N, |u n - c| < ε

-- 1ª demostración
-- ===============

lemma aux
  (ha : limite u a)
  (hb : limite u b)
  : b ≤ a :=
begin
  by_contra h,
  set ε := b - a with hε,
  cases ha (ε/2) (by linarith) with A hA,
  cases hb (ε/2) (by linarith) with B hB,
  set N := max A B with hN,
  have hAN : A ≤ N := le_max_left A B,
  have hBN : B ≤ N := le_max_right A B,
  specialize hA N hAN,
  specialize hB N hBN,
  rw abs_lt at hA hB,
  linarith,
end

example
  (ha : limite u a)
  (hb : limite u b)
  : a = b :=
le_antisymm (aux hb ha) (aux ha hb)

-- 2ª demostración
-- ===============

example
  (ha : limite u a)
  (hb : limite u b)
```

```
  : a = b :=
begin
  by_contra h,
  wlog hab : a < b,
  { have : a < b ∨ a = b ∨ b < a := lt_trichotomy a b,
    tauto },
  set ε := b - a with hε,
  specialize ha (ε/2),
  have hε2 : ε/2 > 0 := by linarith,
  specialize ha hε2,
  cases ha with A hA,
  cases hb (ε/2) (by linarith) with B hB,
  set N := max A B with hN,
  have hAN : A ≤ N := le_max_left A B,
  have hBN : B ≤ N := le_max_right A B,
  specialize hA N hAN,
  specialize hB N hBN,
  rw abs_lt at hA hB,
  linarith,
end

-- 3ª demostración
-- ===============

example
  (ha : limite u a)
  (hb : limite u b)
  : a = b :=
begin
  by_contra h,
  wlog hab : a < b,
  { have : a < b ∨ a = b ∨ b < a := lt_trichotomy a b,
    tauto },
  set ε := b - a with hε,
  cases ha (ε/2) (by linarith) with A hA,
  cases hb (ε/2) (by linarith) with B hB,
  set N := max A B with hN,
  have hAN : A ≤ N := le_max_left A B,
  have hBN : B ≤ N := le_max_right A B,
  specialize hA N hAN,
  specialize hB N hBN,
  rw abs_lt at hA hB,
  linarith,
end
```

Se puede interactuar con las pruebas anteriores en esta sesión con Lean.

4.13. Límite cuando se suma una constante

```
-- ----------------------------------------------------------------------
-- En Lean, una sucesión u₀, u₁, u₂, ... se puede representar mediante
-- una función (u : ℕ → ℝ) de forma que u(n) es uₙ.
--
-- Se define que a es el límite de la sucesión u, por
--    def limite : (ℕ → ℝ) → ℝ → Prop :=
--    λ u a, ∀ ε > 0, ∃ N, ∀ n ≥ N, |u n - a| < ε
-- donde se usa la notación |x| para el valor absoluto de x
--    notation `|`x`|` := abs x
--
-- Demostrar que si el límite de la sucesión u(i) es a y c ∈ ℝ, entonces
-- el límite de u(i)+c es a+c.
-- ----------------------------------------------------------------------

import data.real.basic
import tactic

variables {u : ℕ → ℝ}
variables {a c : ℝ}

notation `|`x`|` := abs x

def limite : (ℕ → ℝ) → ℝ → Prop :=
λ u c, ∀ ε > 0, ∃ N, ∀ n ≥ N, |u n - c| < ε

-- 1ª demostración
-- ===============

example
  (h : limite u a)
  : limite (λ i, u i + c) (a + c) :=
begin
  intros ε hε,
  dsimp,
  cases h ε hε with k hk,
  use k,
  intros n hn,
  calc |u n + c - (a + c)|
       = |u n - a|          : by norm_num
```

```
   ... < ε                 : hk n hn,
end

-- 2ª demostración
-- ===============

example
  (h : limite u a)
  : limite (λ i, u i + c) (a + c) :=
begin
  intros ε hε,
  dsimp,
  cases h ε hε with k hk,
  use k,
  intros n hn,
  convert hk n hn using 2,
  ring,
end

-- 3ª demostración
-- ===============

example
  (h : limite u a)
  : limite (λ i, u i + c) (a + c) :=
begin
  intros ε hε,
  convert h ε hε,
  by norm_num,
end

-- 4ª demostración
-- ===============

example
  (h : limite u a)
  : limite (λ i, u i + c) (a + c) :=
λ ε hε, (by convert h ε hε; norm_num)
```

Se puede interactuar con las pruebas anteriores en esta sesión con Lean.

4.14. Límite de la suma de sucesiones convergentes

```
-- ---------------------------------------------------------------------
-- En Lean, una sucesión u₀, u₁, u₂, ... se puede representar mediante
-- una función (u : ℕ → ℝ) de forma que u(n) es uₙ.
--
-- Se define que a es el límite de la sucesión u, por
--    def limite : (ℕ → ℝ) → ℝ → Prop :=
--    λ u a, ∀ ε > 0, ∃ N, ∀ n ≥ N, |u n - a| < ε
-- donde se usa la notación |x| para el valor absoluto de x
--    notation `|`x`|` := abs x
--
-- Demostrar que el límite de la suma de dos sucesiones convergentes es
-- la suma de los límites de dichas sucesiones.
-- ---------------------------------------------------------------------

import data.real.basic

variables (u v : ℕ → ℝ)
variables (a b : ℝ)

notation `|`x`|` := abs x

def limite : (ℕ → ℝ) → ℝ → Prop :=
λ u c, ∀ ε > 0, ∃ N, ∀ n ≥ N, |u n - c| < ε

-- 1ª demostración
-- ===============

example
  (hu : limite u a)
  (hv : limite v b)
  : limite (u + v) (a + b) :=
begin
  intros ε hε,
  have hε2 : 0 < ε / 2,
    { linarith },
  cases hu (ε / 2) hε2 with Nu hNu,
  cases hv (ε / 2) hε2 with Nv hNv,
  clear hu hv hε2 hε,
  use max Nu Nv,
  intros n hn,
  have hn₁ : n ≥ Nu,
```

```
    { exact le_of_max_le_left hn },
  specialize hNu n hn₁,
  have hn₂ : n ≥ Nv,
    { exact le_of_max_le_right hn },
  specialize hNv n hn₂,
  clear hn hn₁ hn₂ Nu Nv,
  calc |(u + v) n - (a + b)|
       = |(u n + v n) - (a + b)|   : by refl
   ... = |(u n - a) + (v n -  b)| : by {congr, ring}
   ... ≤ |u n - a| + |v n -  b|    : by apply abs_add
   ... < ε / 2 + ε / 2             : by linarith
   ... = ε                         : by apply add_halves,
end

-- 2ª demostración
-- ===============

example
  (hu : limite u a)
  (hv : limite v b)
  : limite (u + v) (a + b) :=
begin
  intros ε hε,
  cases hu (ε/2) (by linarith) with Nu hNu,
  cases hv (ε/2) (by linarith) with Nv hNv,
  use max Nu Nv,
  intros n hn,
  have hn₁ : n ≥ Nu := le_of_max_le_left hn,
  specialize hNu n hn₁,
  have hn₂ : n ≥ Nv := le_of_max_le_right hn,
  specialize hNv n hn₂,
  calc |(u + v) n - (a + b)|
       = |(u n + v n) - (a + b)|   : by refl
   ... = |(u n - a) + (v n -  b)| : by {congr, ring}
   ... ≤ |u n - a| + |v n -  b|    : by apply abs_add
   ... < ε / 2 + ε / 2             : by linarith
   ... = ε                         : by apply add_halves,
end

-- 3ª demostración
-- ===============

lemma max_ge_iff
  {α : Type*}
  [linear_order α]
```

```
  {p q r : α}
  : r ≥ max p q  ↔ r ≥ p ∧ r ≥ q :=
max_le_iff

example
  (hu : limite u a)
  (hv : limite v b)
  : limite (u + v) (a + b) :=
begin
  intros ε hε,
  cases hu (ε/2) (by linarith) with Nu hNu,
  cases hv (ε/2) (by linarith) with Nv hNv,
  use max Nu Nv,
  intros n hn,
  cases max_ge_iff.mp hn with hn₁ hn₂,
  have cota₁ : |u n - a| < ε/2 := hNu n hn₁,
  have cota₂ : |v n - b| < ε/2 := hNv n hn₂,
  calc |(u + v) n - (a + b)|
       = |u n + v n - (a + b)|   : by rfl
   ... = |(u n - a) + (v n - b)| : by { congr, ring }
   ... ≤ |u n - a| + |v n - b|   : by apply abs_add
   ... < ε                       : by linarith,
end

-- 4ª demostración
-- ===============

example
  (hu : limite u a)
  (hv : limite v b)
  : limite (u + v) (a + b) :=
begin
  intros ε hε,
  cases hu (ε/2) (by linarith) with Nu hNu,
  cases hv (ε/2) (by linarith) with Nv hNv,
  use max Nu Nv,
  intros n hn,
  cases max_ge_iff.mp hn with hn₁ hn₂,
  calc |(u + v) n - (a + b)|
       = |u n + v n - (a + b)|   : by refl
   ... = |(u n - a) + (v n - b)| : by { congr, ring }
   ... ≤ |u n - a| + |v n - b|   : by apply abs_add
   ... < ε/2 + ε/2               : add_lt_add (hNu n hn₁) (hNv n hn₂)
   ... = ε                       : by simp
end
```

```
-- 5ª demostración
-- ===============

example
  (hu : limite u a)
  (hv : limite v b)
  : limite (u + v) (a + b) :=
begin
  intros ε hε,
  cases hu (ε/2) (by linarith) with Nu hNu,
  cases hv (ε/2) (by linarith) with Nv hNv,
  use max Nu Nv,
  intros n hn,
  rw max_ge_iff at hn,
  calc |(u + v) n - (a + b)|
       = |u n + v n - (a + b)|   : by refl
   ... = |(u n - a) + (v n - b)| : by { congr, ring }
   ... ≤ |u n - a| + |v n - b|   : by apply abs_add
   ... < ε                       : by linarith [hNu n (by linarith), hNv n (by linarith)]
end

-- 6ª demostración
-- ===============

example
  (hu : limite u a)
  (hv : limite v b)
  : limite (u + v) (a + b) :=
begin
  intros ε Hε,
  cases hu (ε/2) (by linarith) with L HL,
  cases hv (ε/2) (by linarith) with M HM,
  set N := max L M with hN,
  use N,
  have HLN : N ≥ L := le_max_left _ _,
  have HMN : N ≥ M := le_max_right _ _,
  intros n Hn,
  have H3 : |u n - a| < ε/2 := HL n (by linarith),
  have H4 : |v n - b| < ε/2 := HM n (by linarith),
  calc |(u + v) n - (a + b)|
       = |(u n + v n) - (a + b)|   : by refl
   ... = |(u n - a) + (v n - b)|   : by { congr, ring }
   ... ≤ |(u n - a)| + |(v n - b)| : by apply abs_add
   ... < ε/2 + ε/2                 : by linarith
```

```
   ... = ε                              : by ring
end
```

Se puede interactuar con las pruebas anteriores en esta sesión con Lean.

4.15. Límite multiplicado por una constante

```
-- ---------------------------------------------------------------------
-- En Lean, una sucesión u₀, u₁, u₂, ... se puede representar mediante
-- una función (u : ℕ → ℝ) de forma que u(n) es uₙ.
--
-- Se define que a es el límite de la sucesión u, por
--    def limite : (ℕ → ℝ) → ℝ → Prop :=
--    λ u a, ∀ ε > 0, ∃ N, ∀ n ≥ N, |u n - a| < ε
-- donde se usa la notación |x| para el valor absoluto de x
--    notation `|`x`|` := abs x
--
-- Demostrar que que si el límite de u(i) es a, entonces el de
-- c*u(i) es c*a.
-- ---------------------------------------------------------------------

import data.real.basic
import tactic

variables (u v : ℕ → ℝ)
variables (a c : ℝ)

notation `|`x`|` := abs x

def limite : (ℕ → ℝ) → ℝ → Prop :=
λ u c, ∀ ε > 0, ∃ N, ∀ n ≥ N, |u n - c| < ε

-- 1ª demostración
-- ===============

example
  (h : limite u a)
  : limite (λ n, c * (u n)) (c * a) :=
begin
  by_cases hc : c = 0,
  { subst hc,
    intros ε hε,
    by finish, },
```

```
  { intros ε hε,
    have hc' : 0 < |c| := abs_pos.mpr hc,
    have hεc : 0 < ε / |c| := div_pos hε hc',
    specialize h (ε/|c|) hεc,
    cases h with N hN,
    use N,
    intros n hn,
    specialize hN n hn,
    dsimp only,
    rw ← mul_sub,
    rw abs_mul,
    rw ← lt_div_iff' hc',
    exact hN, }
end

-- 2ª demostración
-- ===============

example
  (h : limite u a)
  : limite (λ n, c * (u n)) (c * a) :=
begin
  by_cases hc : c = 0,
  { subst hc,
    intros ε hε,
    by finish, },
  { intros ε hε,
    have hc' : 0 < |c| := abs_pos.mpr hc,
    have hεc : 0 < ε / |c| := div_pos hε hc',
    specialize h (ε/|c|) hεc,
    cases h with N hN,
    use N,
    intros n hn,
    specialize hN n hn,
    dsimp only,
    calc |c * u n - c * a|
         = |c * (u n - a)| : congr_arg abs (mul_sub c (u n) a).symm
     ... = |c| * |u n - a| : abs_mul c  (u n - a)
     ... < |c| * (ε / |c|) : (mul_lt_mul_left hc').mpr hN
     ... = ε               : mul_div_cancel' ε (ne_of_gt hc') }
end

-- 3ª demostración
-- ===============
```

```
example
  (h : limite u a)
  : limite (λ n, c * (u n)) (c * a) :=
begin
  by_cases hc : c = 0,
  { subst hc,
    intros ε hε,
    by finish, },
  { intros ε hε,
    have hc' : 0 < |c| := by finish,
    have hεc : 0 < ε / |c| := div_pos hε hc',
    cases h (ε/|c|) hεc with N hN,
    use N,
    intros n hn,
    specialize hN n hn,
    dsimp only,
    rw [← mul_sub, abs_mul, ← lt_div_iff' hc'],
    exact hN, }
end
```

Se puede interactuar con las pruebas anteriores en esta sesión con Lean.

4.16. El límite de u es a syss el de u-a es 0

```
-- ----------------------------------------------------------------------
-- En Lean, una sucesión u₀, u₁, u₂, ... se puede representar mediante
-- una función (u : ℕ → ℝ) de forma que u(n) es uₙ.
--
-- Se define que a es el límite de la sucesión u, por
--    def limite : (ℕ → ℝ) → ℝ → Prop :=
--    λ u a, ∀ ε > 0, ∃ N, ∀ n ≥ N, |u n - a| < ε
-- donde se usa la notación |x| para el valor absoluto de x
--    notation `|`x`|` := abs x
--
-- Demostrar que el límite de u(i) es a si y solo si el de u(i)-a es
-- 0.
-- ----------------------------------------------------------------------

import data.real.basic
import tactic

variable  {u : ℕ → ℝ}
variables {a c x : ℝ}
```

```
notation `|`x`|` := abs x

def limite : (ℕ → ℝ) → ℝ → Prop :=
λ u c, ∀ ε > 0, ∃ N, ∀ n ≥ N, |u n - c| < ε

-- 1ª demostración
-- ===============

example
  : limite u a ↔ limite (λ i, u i - a) 0 :=
begin
  rw iff_eq_eq,
  calc limite u a
       = ∀ ε > 0, ∃ N, ∀ n ≥ N, |u n - a| < ε       : rfl
   ... = ∀ ε > 0, ∃ N, ∀ n ≥ N, |(u n - a) - 0| < ε : by simp
   ... = limite (λ i, u i - a) 0                    : rfl,
end

-- 2ª demostración
-- ===============

example
  : limite u a ↔ limite (λ i, u i - a) 0 :=
begin
  split,
  { intros h ε hε,
    convert h ε hε,
    norm_num, },
  { intros h ε hε,
    convert h ε hε,
    norm_num, },
end

-- 3ª demostración
-- ===============

example
  : limite u a ↔ limite (λ i, u i - a) 0 :=
begin
  split;
  { intros h ε hε,
    convert h ε hε,
    norm_num, },
end
```

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

lemma limite_con_suma
  (c : ℝ)
  (h : limite u a)
  : limite (λ i, u i + c) (a + c) :=
λ ε hε, (by convert h ε hε; norm_num)

lemma CNS_limite_con_suma
  (c : ℝ)
  : limite u a ↔ limite (λ i, u i + c) (a + c) :=
begin
  split,
  { apply limite_con_suma },
  { intro h,
    convert limite_con_suma (-c) h; simp, },
end

example
  (u : ℕ → ℝ)
  (a : ℝ)
  : limite u a ↔ limite (λ i, u i - a) 0 :=
begin
  convert CNS_limite_con_suma (-a),
  simp,
end
```

Se puede interactuar con las pruebas anteriores en esta sesión con Lean.

4.17. Producto de sucesiones convergentes a cero

```
-- ----------------------------------------------------------------------
-- En Lean, una sucesión u₀, u₁, u₂, ... se puede representar mediante
-- una función (u : ℕ → ℝ) de forma que u(n) es uₙ.
--
-- Se define que a es el límite de la sucesión u, por
--    def limite : (ℕ → ℝ) → ℝ → Prop :=
--    λ u a, ∀ ε > 0, ∃ N, ∀ n ≥ N, |u n - a| < ε
-- donde se usa la notación |x| para el valor absoluto de x
--    notation `|`x`|` := abs x
```

```
--
-- Demostrar que si las sucesiones u(n) y v(n) convergen a cero,
-- entonces u(n)·v(n) también converge a cero.
-- ---------------------------------------------------------------------

import data.real.basic
import tactic

variables {u v : ℕ → ℝ}
variables {: ℝ}

notation `|`x`|` := abs x

def limite : (ℕ → ℝ) → ℝ → Prop :=
λ u c, ∀ ε > 0, ∃ N, ∀ n ≥ N, |u n - c| < ε

-- 1ª demostración
-- ===============

example
  (hu : limite u 0)
  (hv : limite v 0)
  : limite (u * v) 0 :=
begin
  intros ε hε,
  cases hu ε hε with U hU,
  cases hv 1 zero_lt_one with V hV,
  set N := max U V with hN,
  use N,
  intros n hn,
  specialize hU n (le_of_max_le_left hn),
  specialize hV n (le_of_max_le_right hn),
  rw sub_zero at *,
  calc |(u * v) n|
       = |u n * v n|   : rfl
   ... = |u n| * |v n| : abs_mul (u n) (v n)
   ... < ε * 1         : mul_lt_mul'' hU hV (abs_nonneg (u n)) (abs_nonneg (v n))
   ... = ε             : mul_one ε,
end

-- 2ª demostración
-- ===============

example
  (hu : limite u 0)
```

```
  (hv : limite v 0)
  : limite (u * v) 0 :=
begin
  intros ε hε,
  cases hu ε hε with U hU,
  cases hv 1 (by linarith) with V hV,
  set N := max U V with hN,
  use N,
  intros n hn,
  specialize hU n (le_of_max_le_left hn),
  specialize hV n (le_of_max_le_right hn),
  rw sub_zero at *,
  calc |(u * v) n|
       = |u n * v n|    : rfl
   ... = |u n| * |v n| : abs_mul (u n) (v n)
   ... < ε * 1          : by { apply mul_lt_mul'' hU hV ; simp [abs_nonneg] }
   ... = ε              : mul_one ε,
end

-- 3ª demostración
-- ===============

example
  (hu : limite u 0)
  (hv : limite v 0)
  : limite (u * v) 0 :=
begin
  intros ε hε,
  cases hu ε hε with U hU,
  cases hv 1 (by linarith) with V hV,
  set N := max U V with hN,
  use N,
  intros n hn,
  have hUN : U ≤ N := le_max_left U V,
  have hVN : V ≤ N := le_max_right U V,
  specialize hU n (by linarith),
  specialize hV n (by linarith),
  rw sub_zero at ⊢ hU hV,
  rw pi.mul_apply,
  rw abs_mul,
  convert mul_lt_mul'' hU hV _ _, simp,
  all_goals {apply abs_nonneg},
end
```

Se puede interactuar con las pruebas anteriores en esta sesión con Lean.

4.18. Teorema del emparedado

```
-- ---------------------------------------------------------------------
-- En Lean, una sucesión u₀, u₁, u₂, ... se puede representar mediante
-- una función (u : ℕ → ℝ) de forma que u(n) es uₙ.
--
-- Se define que a es el límite de la sucesión u, por
--    def limite : (ℕ → ℝ) → ℝ → Prop :=
--    λ u a, ∀ ε > 0, ∃ N, ∀ n ≥ N, |u n - a| < ε
-- donde se usa la notación |x| para el valor absoluto de x
--    notation `|`x`|` := abs x
--
-- Demostrar que si para todo n, u(n) ≤ v(n) ≤ w(n) y u(n) tiene el
-- mismo límite que w(n), entonces v(n) también tiene dicho límite.
-- ---------------------------------------------------------------------

import data.real.basic

variables (u v w : ℕ → ℝ)
variable  (a : ℝ)

notation `|`x`|` := abs x

def limite : (ℕ → ℝ) → ℝ → Prop :=
λ u c, ∀ ε > 0, ∃ N, ∀ n ≥ N, |u n - c| ≤ ε

-- Nota. En la demostración se usará el siguiente lema:
lemma max_ge_iff
  {p q r : ℕ}
  : r ≥ max p q ↔ r ≥ p ∧ r ≥ q :=
max_le_iff

-- 1ª demostración
-- ===============

example
  (hu : limite u a)
  (hw : limite w a)
  (h : ∀ n, u n ≤ v n)
  (h' : ∀ n, v n ≤ w n) :
  limite v a :=
begin
  intros ε hε,
  cases hu ε hε with N hN, clear hu,
```

```
  cases hw ε hε with N' hN', clear hw hε,
  use max N N',
  intros n hn,
  rw max_ge_iff at hn,
  specialize hN n hn.1,
  specialize hN' n hn.2,
  specialize h n,
  specialize h' n,
  clear hn,
  rw abs_le at *,
  split,
  { calc -ε
         ≤ u n - a : hN.1
     ... ≤ v n - a : by linarith, },
  { calc v n - a
         ≤ w n - a : by linarith
     ... ≤ ε       : hN'.2, },
end

-- 2ª demostración
example
  (hu : limite u a)
  (hw : limite w a)
  (h : ∀ n, u n ≤ v n)
  (h' : ∀ n, v n ≤ w n) :
  limite v a :=
begin
  intros ε hε,
  cases hu ε hε with N hN, clear hu,
  cases hw ε hε with N' hN', clear hw hε,
  use max N N',
  intros n hn,
  rw max_ge_iff at hn,
  specialize hN n (by linarith),
  specialize hN' n (by linarith),
  specialize h n,
  specialize h' n,
  rw abs_le at *,
  split,
  { linarith, },
  { linarith, },
end

-- 3ª demostración
example
```

```
  (hu : limite u a)
  (hw : limite w a)
  (h : ∀ n, u n ≤ v n)
  (h' : ∀ n, v n ≤ w n) :
  limite v a :=
begin
  intros ε hε,
  cases hu ε hε with N hN, clear hu,
  cases hw ε hε with N' hN', clear hw hε,
  use max N N',
  intros n hn,
  rw max_ge_iff at hn,
  specialize hN n (by linarith),
  specialize hN' n (by linarith),
  specialize h n,
  specialize h' n,
  rw abs_le at *,
  split ; linarith,
end

-- 4ª demostración
example
  (hu : limite u a)
  (hw : limite w a)
  (h : ∀ n, u n ≤ v n)
  (h' : ∀ n, v n ≤ w n) :
  limite v a :=
assume ε,
assume hε : ε > 0,
exists.elim (hu ε hε)
  ( assume N,
    assume hN : ∀ (n : ℕ), n ≥ N → |u n - a| ≤ ε,
    exists.elim (hw ε hε)
      ( assume N',
        assume hN' : ∀ (n : ℕ), n ≥ N' → |w n - a| ≤ ε,
        show ∃ N, ∀ n, n ≥ N → |v n - a| ≤ ε, from
          exists.intro (max N N')
            ( assume n,
              assume hn : n ≥ max N N',
              have h1 : n ≥ N ∧ n ≥ N',
                from max_ge_iff.mp hn,
              have h2 : -ε ≤ v n - a,
                { have h2a : |u n - a| ≤ ε,
                    from hN n h1.1,
                  calc -ε
```

```
                  ≤ u n - a : and.left (abs_le.mp h2a)
              ... ≤ v n - a : by linarith [h n], },
         have h3 : v n - a ≤ ε,
           { have h3a : |w n - a| ≤ ε,
               from hN' n h1.2,
             calc v n - a
                  ≤ w n - a : by linarith [h' n]
              ... ≤ ε       : and.right (abs_le.mp h3a), },
         show |v n - a| ≤ ε,
           from abs_le.mpr (and.intro h2 h3))))
```

Se puede interactuar con las pruebas anteriores en esta sesión con Lean.

4.19. La composición de crecientes es creciente

```
-- ---------------------------------------------------------------------
-- Se dice que una función f de ℝ en ℝ es creciente https://bit.ly/2UShggL
-- si para todo x e y tales que x ≤ y se tiene que f(x) ≤ f(y).
--
-- En Lean que f sea creciente se representa por `monotone f`.
--
-- Demostrar que la composición de dos funciones crecientes es una
-- función creciente.
-- ---------------------------------------------------------------------

import data.real.basic

variables (f g : ℝ → ℝ)

-- 1ª demostración
example
  (hf : monotone f)
  (hg : monotone g)
  : monotone (g ∘ f) :=
begin
  intros x y hxy,
  calc (g ∘ f) x
       = g (f x)   : rfl
   ... ≤ g (f y)   : hg (hf hxy)
   ... = (g ∘ f) y : rfl,
end
```

```
-- 2ª demostración
example
  (hf : monotone f)
  (hg : monotone g)
  : monotone (g ∘ f) :=
begin
  unfold monotone at *,
  intros x y h,
  unfold function.comp,
  apply hg,
  apply hf,
  exact h,
end

-- 3ª demostración
example
  (hf : monotone f)
  (hg : monotone g)
  : monotone (g ∘ f) :=
begin
  intros x y h,
  apply hg,
  apply hf,
  exact h,
end

-- 4ª demostración
example
  (hf : monotone f)
  (hg : monotone g)
  : monotone (g ∘ f) :=
begin
  intros x xy h,
  apply hg,
  exact hf h,
end

-- 5ª demostración
example
  (hf : monotone f)
  (hg : monotone g)
  : monotone (g ∘ f) :=
begin
  intros x y h,
```

```
  exact hg (hf h),
end

-- 6ª demostración
example
  (hf : monotone f)
  (hg : monotone g)
  : monotone (g ∘ f) :=
λ x y h, hg (hf h)

-- 7ª demostración
example
  (hf : monotone f)
  (hg : monotone g)
  : monotone (g ∘ f) :=
begin
  intros x y h,
  specialize hf h,
  exact hg hf,
end

-- 8ª demostración
example
  (hf : monotone f)
  (hg : monotone g)
  : monotone (g ∘ f) :=
assume x y,
assume h1 : x ≤ y,
have h2 : f x ≤ f y,
  from hf h1,
show (g ∘ f) x ≤ (g ∘ f) y, from
  calc (g ∘ f) x
       = g (f x)   : rfl
   ... ≤ g (f y)   : hg h2
   ... = (g ∘ f) y : by refl

-- 9ª demostración
example
  (hf : monotone f)
  (hg : monotone g)
  : monotone (g ∘ f) :=
-- by hint
by tauto

-- 10ª demostración
```

```
example
  (hf : monotone f)
  (hg : monotone g)
  : monotone (g ∘ f) :=
-- by library_search
monotone.comp hg hf
```

Se puede interactuar con las pruebas anteriores en esta sesión con Lean.

4.20. La composición de una función creciente y una decreciente es decreciente

```
-- ---------------------------------------------------------------------
-- Sea una función f de ℝ en ℝ. Se dice que f es creciente
-- https://bit.ly/2UShggL si para todo x e y tales que x ≤ y se tiene
-- que f(x) ≤ f(y). Se dice que f es decreciente si para todo x e y
-- tales que x ≤ y se tiene que f(x) ≥ f(y).
--
-- Demostrar que si f es creciente y g es decreciente, entonces (g ∘ f)
-- es decreciente.
-- ---------------------------------------------------------------------

import data.real.basic

variables (f g : ℝ → ℝ)

def creciente (f : ℝ → ℝ) : Prop :=
∀ {x y}, x ≤ y → f x ≤ f y

def decreciente (f : ℝ → ℝ) : Prop :=
∀ {x y}, x ≤ y → f x ≥ f y

-- 1ª demostración
example
  (hf : creciente f)
  (hg : decreciente g)
  : decreciente (g ∘ f) :=
begin
  intros x y hxy,
  calc (g ∘ f) x
       = g (f x)   : rfl
   ... ≥ g (f y)   : hg (hf hxy)
```

```
    ... = (g ∘ f) y : rfl,
end

-- 2ª demostración
example
  (hf : creciente f)
  (hg : decreciente g)
  : decreciente (g ∘ f) :=
begin
  unfold creciente decreciente at *,
  intros x y h,
  unfold function.comp,
  apply hg,
  apply hf,
  exact h,
end

-- 3ª demostración
example
  (hf : creciente f)
  (hg : decreciente g)
  : decreciente (g ∘ f) :=
begin
  intros x y h,
  apply hg,
  apply hf,
  exact h,
end

-- 4ª demostración
example
  (hf : creciente f)
  (hg : decreciente g)
  : decreciente (g ∘ f) :=
begin
  intros x y h,
  apply hg,
  exact hf h,
end

-- 5ª demostración
example
  (hf : creciente f)
  (hg : decreciente g)
  : decreciente (g ∘ f) :=
```

```
begin
  intros x y h,
  exact hg (hf h),
end

-- 6ª demostración
example
  (hf : creciente f)
  (hg : decreciente g)
  : decreciente (g ∘ f) :=
λ x y h, hg (hf h)

-- 7ª demostración
example
  (hf : creciente f)
  (hg : decreciente g)
  : decreciente (g ∘ f) :=
assume x y,
assume h : x ≤ y,
have h1 : f x ≤ f y,
  from hf h,
show (g ∘ f) x ≥ (g ∘ f) y,
  from hg h1

-- 8ª demostración
example
  (hf : creciente f)
  (hg : decreciente g)
  : decreciente (g ∘ f) :=
assume x y,
assume h : x ≤ y,
show (g ∘ f) x ≥ (g ∘ f) y,
  from hg (hf h)

-- 9ª demostración
example
  (hf : creciente f)
  (hg : decreciente g)
  : decreciente (g ∘ f) :=
λ x y h, hg (hf h)

-- 10ª demostración
example
  (hf : creciente f)
  (hg : decreciente g)
```

```
  : decreciente (g ∘ f) :=
-- by hint
by tauto
```

4.21. Una función creciente e involutiva es la identidad

```
-- ---------------------------------------------------------------------
-- Sea una función f de ℝ en ℝ.
-- + Se dice que f es creciente si para todo x e y tales que x ≤ y se
--   tiene que f(x) ≤ f(y).
-- + Se dice que f es involutiva si para todo x se tiene que f(f(x)) = x.
--
-- En Lean que f sea creciente se representa por `monotone f` y que sea
-- involutiva por `involutive f`
--
-- Demostrar que si f es creciente e involutiva, entonces f es la
-- identidad.
-- ---------------------------------------------------------------------

import data.real.basic
open function

variable (f : ℝ → ℝ)

-- 1ª demostración
example
  (hc : monotone f)
  (hi : involutive f)
  : f = id :=
begin
  unfold monotone involutive at *,
  funext,
  unfold id,
  cases (le_total (f x) x) with h1 h2,
  { apply antisymm h1,
    have h3 : f (f x) ≤ f x,
      { apply hc,
        exact h1, },
    rwa hi at h3, },
  { apply antisymm _ h2,
```

```
    have h4 : f x ≤ f (f x),
      { apply hc,
        exact h2, },
    rwa hi at h4, },
end

-- 2ª demostración
example
  (hc : monotone f)
  (hi : involutive f)
  : f = id :=
begin
  funext,
  cases (le_total (f x) x) with h1 h2,
  { apply antisymm h1,
    have h3 : f (f x) ≤ f x := hc h1,
    rwa hi at h3, },
  { apply antisymm _ h2,
    have h4 : f x ≤ f (f x) := hc h2,
    rwa hi at h4, },
end

-- 3ª demostración
example
  (hc : monotone f)
  (hi : involutive f)
  : f = id :=
begin
  funext,
  cases (le_total (f x) x) with h1 h2,
  { apply antisymm h1,
    calc x
         = f (f x) : (hi x).symm
     ... ≤ f x     : hc h1 },
  { apply antisymm _ h2,
    calc f x
         ≤ f (f x) : hc h2
     ... = x       : hi x },
end
```

Se puede interactuar con las pruebas anteriores en esta sesión con Lean.

4.22. Si ‘f x ≤ f y → x ≤ y‘, entonces f es inyectiva

```
-- ---------------------------------------------------------------------
-- Sea f una función de ℝ en ℝ tal que
--    ∀ x y, f(x) ≤ f(y) → x ≤ y
-- Demostrar que f es inyectiva.
-- ---------------------------------------------------------------------

import data.real.basic
open function

variable (f : ℝ → ℝ)

-- 1ª demostración
example
  (h : ∀ {x y}, f x ≤ f y → x ≤ y)
  : injective f :=
begin
  intros x y hxy,
  apply le_antisymm,
  { apply h,
    exact le_of_eq hxy, },
  { apply h,
    exact ge_of_eq hxy, },
end

-- 2ª demostración
example
  (h : ∀ {x y}, f x ≤ f y → x ≤ y)
  : injective f :=
begin
  intros x y hxy,
  apply le_antisymm,
  { exact h (le_of_eq hxy), },
  { exact h (ge_of_eq hxy), },
end

-- 3ª demostración
example
  (h : ∀ {x y}, f x ≤ f y → x ≤ y)
  : injective f :=
λ x y hxy, le_antisymm (h hxy.le) (h hxy.ge)
```

Se puede interactuar con las pruebas anteriores en esta sesión con Lean.

4.23. Los supremos de las sucesiones crecientes son sus límites

```
-- ----------------------------------------------------------------------
-- Sea u una sucesión creciente. Demostrar que si M es un supremo de u,
-- entonces el límite de u es M.
-- ----------------------------------------------------------------------

import data.real.basic

variable (u : ℕ → ℝ)
variable (M : ℝ)

notation `|`x`|` := abs x

-- (limite u c) expresa que el límite de u es c.
def limite (u : ℕ → ℝ) (c : ℝ) :=
  ∀ ε > 0, ∃ N, ∀ n ≥ N, |u n - c| ≤ ε

-- (supremo u M) expresa que el supremo de u es M.
def supremo (u : ℕ → ℝ) (M : ℝ) :=
  (∀ n, u n ≤ M) ∧ ∀ ε > 0, ∃ n₀, u n₀ ≥ M - ε

-- 1ª demostración
example
  (hu : monotone u)
  (hM : supremo u M)
  : limite u M :=
begin
  -- unfold limite,
  intros ε hε,
  -- unfold supremo at h,
  cases hM with hM₁ hM₂,
  cases hM₂ ε hε with n₀ hn₀,
  use n₀,
  intros n hn,
  rw abs_le,
  split,
  { -- unfold monotone at h',
    specialize hu hn,
```

```
    calc -ε
        = (M - ε) - M : by ring
    ... ≤ u n₀ - M    : sub_le_sub_right hn₀ M
    ... ≤ u n - M     : sub_le_sub_right hu M },
  { calc u n - M
        ≤ M - M       : sub_le_sub_right (hM₁ n) M
    ... = 0           : sub_self M
    ... ≤ ε           : le_of_lt hε, },
end

-- 2ª demostración
example
  (hu : monotone u)
  (hM : supremo u M)
  : limite u M :=
begin
  intros ε hε,
  cases hM with hM₁ hM₂,
  cases hM₂ ε hε with n₀ hn₀,
  use n₀,
  intros n hn,
  rw abs_le,
  split,
  { linarith [hu hn] },
  { linarith [hM₁ n] },
end

-- 3ª demostración
example
  (hu : monotone u)
  (hM : supremo u M)
  : limite u M :=
begin
  intros ε hε,
  cases hM with hM₁ hM₂,
  cases hM₂ ε hε with n₀ hn₀,
  use n₀,
  intros n hn,
  rw abs_le,
  split ; linarith [hu hn, hM₁ n],
end

-- 4ª demostración
example
  (hu : monotone u)
```

```
  (hM : supremo u M)
  : limite u M :=
assume ε,
assume hε : ε > 0,
have hM₁ : ∀ (n : ℕ), u n ≤ M,
  from hM.left,
have hM₂ : ∀ (ε : ℝ), ε > 0 → (∃ (n₀ : ℕ), u n₀ ≥ M - ε),
  from hM.right,
exists.elim (hM₂ ε hε)
  ( assume n₀,
    assume hn₀ : u n₀ ≥ M - ε,
    have h1 : ∀ n, n ≥ n₀ → |u n - M| ≤ ε,
      { assume n,
        assume hn : n ≥ n₀,
        have h2 : -ε ≤ u n - M,
          { have h3 : u n₀ ≤ u n,
              from hu hn,
            calc -ε
                 = (M - ε) - M : by ring
             ... ≤ u n₀ - M    : sub_le_sub_right hn₀ M
             ... ≤ u n - M     : sub_le_sub_right h3 M },
        have h4 : u n - M ≤ ε,
          { calc u n - M
                 ≤ M - M       : sub_le_sub_right (hM₁ n) M
             ... = 0           : sub_self M
             ... ≤ ε           : le_of_lt hε },
        show |u n - M| ≤ ε,
          from abs_le.mpr (and.intro h2 h4) },
    show ∃ N, ∀ n, n ≥ N → |u n - M| ≤ ε,
      from exists.intro n₀ h1)
```

Se puede interactuar con las pruebas anteriores en esta sesión con Lean.

4.24. Un número es par syss lo es su cuadrado

```
-- ----------------------------------------------------------------------
-- Demostrar que un número es par si y solo si lo es su cuadrado.
-- ----------------------------------------------------------------------

import data.int.parity
import tactic
open int
```

```
variable (n : ℤ)

-- 1ª demostración
example :
  even (n^2) ↔ even n :=
begin
  split,
  { contrapose,
    rw ← odd_iff_not_even,
    rw ← odd_iff_not_even,
    unfold odd,
    intro h,
    cases h with k hk,
    use 2*k*(k+1),
    rw hk,
    ring, },
  { unfold even,
    intro h,
    cases h with k hk,
    use 2*k^2,
    rw hk,
    ring, },
end

-- 2ª demostración
example :
  even (n^2) ↔ even n :=
begin
  split,
  { contrapose,
    rw ← odd_iff_not_even,
    rw ← odd_iff_not_even,
    rintro ⟨k, rfl⟩,
    use 2*k*(k+1),
    ring, },
  { rintro ⟨k, rfl⟩,
    use 2*k^2,
    ring, },
end

-- 3ª demostración
example :
  even (n^2) ↔ even n :=
iff.intro
```

```
  ( have h : ¬even n → ¬even (n^2),
      { assume h1 : ¬even n,
        have h2 : odd n,
          from odd_iff_not_even.mpr h1,
        have h3: odd (n^2), from
          exists.elim h2
            ( assume k,
              assume hk : n = 2*k+1,
              have h4 : n^2 = 2*(2*k*(k+1))+1, from
                calc  n^2
                    = (2*k+1)^2       : by rw hk
                ... = 4*k^2+4*k+1     : by ring
                ... = 2*(2*k*(k+1))+1 : by ring,
              show odd (n^2),
                from exists.intro (2*k*(k+1)) h4),
        show ¬even (n^2),
          from odd_iff_not_even.mp h3 },
    show even (n^2) → even n,
      from not_imp_not.mp h )
  ( assume h1 : even n,
    show even (n^2), from
      exists.elim h1
        ( assume k,
          assume hk : n = 2*k ,
          have h2 : n^2 = 2*(2*k^2), from
            calc  n^2
                = (2*k)^2   : by rw hk
            ... = 2*(2*k^2) : by ring,
          show even (n^2),
            from exists.intro (2*k^2) h2 ))

-- 4ª demostración
example :
  even (n^2) ↔ even n :=
calc even (n^2)
     ↔ even (n * n)       : iff_of_eq (congr_arg even (sq n))
 ... ↔ (even n ∨ even n) : int.even_mul
 ... ↔ even n             : or_self (even n)

-- 5ª demostración
example :
  even (n^2) ↔ even n :=
calc even (n^2)
```

```
     ↔ even (n * n)      : by ring_nf
 ... ↔ (even n ∨ even n) : int.even_mul
 ... ↔ even n            : by simp

-- 6ª demostración
example :
  even (n^2) ↔ even n :=
begin
  split,
  { contrapose,
    intro h,
    rw ← odd_iff_not_even at *,
    cases h with k hk,
    use 2*k*(k+1),
    calc n^2
         = (2*k+1)^2       : by rw hk
     ... = 4*k^2+4*k+1     : by ring
     ... = 2*(2*k*(k+1))+1 : by ring, },
  { intro h,
    cases h with k hk,
    use 2*k^2,
    calc n^2
         = (2*k)^2   : by rw hk
     ... = 2*(2*k^2) : by ring, },
end
```

Se puede interactuar con las pruebas anteriores en esta sesión con Lean.

4.25. Acotación de sucesiones convergente

```
-- ----------------------------------------------------------------------
-- Demostrar que si u es una sucesión convergente, entonces está
-- acotada; es decir,
--     ∃ k b. ∀n≥k. ¦u n¦ ≤ b
-- -----------------------------------------------------------------------

import data.real.basic

variable {u : ℕ → ℝ}
variable {a : ℝ}

notation `|`x`|` := abs x
```

```
-- (limite u c) expresa que el límite de u es c.
def limite (u : ℕ → ℝ) (c : ℝ) :=
  ∀ ε > 0, ∃ k, ∀ n ≥ k, |u n - c| ≤ ε

-- (convergente u) expresa que u es convergente.
def convergente (u : ℕ → ℝ) :=
  ∃ a, limite u a

-- 1ª demostración
example
  (h : convergente u)
  : ∃ k b, ∀ n, n ≥ k → |u n| ≤ b :=
begin
  cases h with a ua,
  cases ua 1 zero_lt_one with k h,
  use [k, 1 + |a|],
  intros n hn,
  specialize h n hn,
  calc |u n|
       = |u n - a + a|    : congr_arg abs (eq_add_of_sub_eq rfl)
   ... ≤ |u n - a| + |a| : abs_add (u n - a) a
   ... ≤ 1 + |a|          : add_le_add_right h _
end

-- 2ª demostración
example
  (h : convergente u)
  : ∃ k b, ∀ n, n ≥ k → |u n| ≤ b :=
begin
  cases h with a ua,
  cases ua 1 zero_lt_one with k h,
  use [k, 1 + |a|],
  intros n hn,
  specialize h n hn,
  calc |u n|
       = |u n - a + a|    : by ring_nf
   ... ≤ |u n - a| + |a| : abs_add (u n - a) a
   ... ≤ 1 + |a|          : by linarith,
end
```

Se puede interactuar con las pruebas anteriores en esta sesión con Lean.

4.26. La paradoja del barbero

```
-- ----------------------------------------------------------------------
-- Demostrar la paradoja del barbero https://bit.ly/3eWyvVw es decir,
-- que no existe un hombre que afeite a todos los que no se afeitan a sí
-- mismo y sólo a los que no se afeitan a sí mismo.
-- ----------------------------------------------------------------------

import tactic

variable (Hombre : Type)
variable (afeita : Hombre → Hombre → Prop)

-- 1ª demostración
example :
  ¬(∃ x : Hombre, ∀ y : Hombre, afeita x y ↔ ¬ afeita y y) :=
begin
  intro h,
  cases h with b hb,
  specialize hb b,
  by_cases (afeita b b),
  { apply absurd h,
    exact hb.mp h, },
  { apply h,
    exact hb.mpr h, },
end

-- 2ª demostración
example :
  ¬(∃ x : Hombre, ∀ y : Hombre, afeita x y ↔ ¬ afeita y y) :=
begin
  intro h,
  cases h with b hb,
  specialize hb b,
  by_cases (afeita b b),
  { exact (hb.mp h) h, },
  { exact h (hb.mpr h), },
end

-- 3ª demostración
example :
  ¬(∃ x : Hombre, ∀ y : Hombre, afeita x y ↔ ¬ afeita y y) :=
begin
  intro h,
```

```
  cases h with b hb,
  specialize hb b,
  by itauto,
end

-- 4ª demostración
example :
  ¬ (∃ x : Hombre,  ∀ y : Hombre, afeita x y ↔ ¬ afeita y y ) :=
begin
  rintro ⟨b, hb⟩,
  exact (iff_not_self (afeita b b)).mp (hb b),
end

-- 5ª demostración
example :
  ¬ (∃ x : Hombre,  ∀ y : Hombre, afeita x y ↔ ¬ afeita y y ) :=
λ ⟨b, hb⟩, (iff_not_self (afeita b b)).mp (hb b)
```

Se puede interactuar con las pruebas anteriores en esta sesión con Lean.

4.27. Propiedad de la densidad de los reales

```
-- ---------------------------------------------------------------------
-- Sean x, y números reales tales que
--    ∀ z, y < z → x ≤ z
-- Demostrar que x ≤ y.
-- ---------------------------------------------------------------------

import data.real.basic

variables {x y : ℝ}

-- 1ª demostración
example
  (h : ∀ z, y < z → x ≤ z) :
  x ≤ y :=
begin
  apply le_of_not_gt,
  intro hxy,
  cases (exists_between hxy) with a ha,
  apply (lt_irrefl a),
  calc a
      < x : ha.2
```

```
    ... ≤ a : h a ha.1,
end

-- 2ª demostración
example
  (h : ∀ z, y < z → x ≤ z) :
  x ≤ y :=
begin
  apply le_of_not_gt,
  intro hxy,
  cases (exists_between hxy) with a ha,
  apply (lt_irrefl a),
  exact lt_of_lt_of_le ha.2 (h a ha.1),
end

-- 3ª demostración
example
  (h : ∀ z, y < z → x ≤ z) :
  x ≤ y :=
begin
  apply le_of_not_gt,
  intro hxy,
  cases (exists_between hxy) with a ha,
  exact (lt_irrefl a) (lt_of_lt_of_le ha.2 (h a ha.1)),
end

-- 3ª demostración
example
  (h : ∀ z, y < z → x ≤ z) :
  x ≤ y :=
begin
  apply le_of_not_gt,
  intro hxy,
  rcases (exists_between hxy) with ⟨a, ha⟩,
  exact (lt_irrefl a) (lt_of_lt_of_le ha.2 (h a ha.1)),
end

-- 4ª demostración
example
  (h : ∀ z, y < z → x ≤ z) :
  x ≤ y :=
begin
  apply le_of_not_gt,
  intro hxy,
  rcases (exists_between hxy) with ⟨a, hya, hax⟩,
```

```
  exact (lt_irrefl a) (lt_of_lt_of_le hax (h a hya)),
end

-- 5ª demostración
example
  (h : ∀ z, y < z → x ≤ z) :
  x ≤ y :=
le_of_not_gt (λ hxy,
  let ⟨a, hya, hax⟩ := exists_between hxy in
  lt_irrefl a (lt_of_lt_of_le hax (h a hya)))

-- 6ª demostración
example
  (h : ∀ z, y < z → x ≤ z) :
  x ≤ y :=
le_of_forall_le_of_dense h
```

Se puede interactuar con las pruebas anteriores en esta sesión con Lean.

4.28. Propiedad cancelativa del producto de números naturales

```
-- ---------------------------------------------------------------------
-- Sean k, m, n números naturales. Demostrar que
--    k * m = k * n ↔ m = n ∨ k = 0
-- ---------------------------------------------------------------------

import data.nat.basic
open nat

variables {k m n : ℕ}

-- Para que no use la notación con puntos
set_option pp.structure_projections false

-- 1ª demostración
example :
  k * m = k * n ↔ m = n ∨ k = 0 :=
begin
  have h1: k ≠ 0 → k * m = k * n → m = n,
    { induction n with n HI generalizing m,
      { by finish, },
```

```
      { cases m,
        { by finish, },
        { intros hk hS,
          congr,
          apply HI hk,
          rw mul_succ at hS,
          rw mul_succ at hS,
          exact add_right_cancel hS, }}},
  by finish,
end

-- 2ª demostración
example :
  k * m = k * n ↔ m = n ∨ k = 0 :=
begin
  have h1: k ≠ 0 → k * m = k * n → m = n,
    { induction n with n HI generalizing m,
      { by finish, },
      { cases m,
        { by finish, },
        { intros hk hS,
          congr,
          apply HI hk,
          simp only [mul_succ] at hS,
          exact add_right_cancel hS, }}},
  by finish,
end

-- 3ª demostración
example :
  k * m = k * n ↔ m = n ∨ k = 0 :=
begin
  have h1: k ≠ 0 → k * m = k * n → m = n,
    { induction n with n HI generalizing m,
      { by finish, },
      { cases m,
        { by finish, },
        { by finish, }}},
  by finish,
end

-- 4ª demostración
example :
  k * m = k * n ↔ m = n ∨ k = 0 :=
begin
```

```
  have h1: k ≠ 0 → k * m = k * n → m = n,
    { induction n with n HI generalizing m,
      { by finish, },
      { cases m; by finish }},
  by finish,
end

-- 5ª demostración
example :
  k * m = k * n ↔ m = n ∨ k = 0 :=
begin
  have h1: k ≠ 0 → k * m = k * n → m = n,
    { induction n with n HI generalizing m ; by finish },
  by finish,
end

-- 5ª demostración
example :
  k * m = k * n ↔ m = n ∨ k = 0 :=
begin
  by_cases hk : k = 0,
  { by simp, },
  { rw mul_right_inj' hk,
    by tauto, },
end

-- 6ª demostración
example :
  k * m = k * n ↔ m = n ∨ k = 0 :=
mul_eq_mul_left_iff

-- 7ª demostración
example :
  k * m = k * n ↔ m = n ∨ k = 0 :=
by simp
```

Se puede interactuar con las pruebas anteriores en esta sesión con Lean.

4.29. Límite de sucesión menor que otra sucesión

```
-- ----------------------------------------------------------------------
-- En Lean, una sucesión u₀, u₁, u₂, ... se puede representar mediante
-- una función (u : ℕ → ℝ) de forma que u(n) es uₙ.
--
-- Se define que a es el límite de la sucesión u, por
--    def limite : (ℕ → ℝ) → ℝ → Prop :=
--    λ u a, ∀ ε > 0, ∃ N, ∀ n ≥ N, |u n - a| < ε
-- donde se usa la notación |x| para el valor absoluto de x
--    notation `|`x`|` := abs x
--
-- Demostrar que si uₙ → a, vₙ → c y uₙ ≤ vₙ para todo n, entonces
-- a ≤ c.
-- ----------------------------------------------------------------------

import data.real.basic
import tactic

variables (u v : ℕ → ℝ)
variables (a c : ℝ)

notation `|`x`|` := abs x

def limite (u : ℕ → ℝ) (c : ℝ) :=
∀ ε > 0, ∃ N, ∀ n ≥ N, |u n - c| < ε

-- 1ª demostración
example
  (hu : limite u a)
  (hv : limite v c)
  (hle : ∀ n, u n ≤ v n)
  : a ≤ c :=
begin
  apply le_of_not_lt,
  intro hlt,
  set ε := (a - c) /2 with hεac,
  have hε : 0 < ε :=
    half_pos (sub_pos.mpr hlt),
  cases hu ε hε with Nu HNu,
  cases hv ε hε with Nv HNv,
  let N := max Nu Nv,
  have HNu' : Nu ≤ N := le_max_left Nu Nv,
```

```
  have HNv' : Nv ≤ N := le_max_right Nu Nv,
  have Ha : |u N - a| < ε := HNu N HNu',
  have Hc : |v N - c| < ε := HNv N HNv',
  have HN : u N ≤ v N := hle N,
  apply lt_irrefl (a - c),
  calc a - c
       = (a - u N) + (u N - c)    : by ring
   ... ≤ (a - u N) + (v N - c)    : by simp [HN]
   ... ≤ |(a - u N) + (v N - c)| : le_abs_self ((a - u N) + (v N - c))
   ... ≤ |a - u N| + |v N - c|    : abs_add (a - u N) (v N - c)
   ... = |u N - a| + |v N - c|    : by simp only [abs_sub]
   ... < ε + ε                    : add_lt_add Ha Hc
   ... = a - c                    : add_halves (a - c),
end

-- 2ª demostración
example
  (hu : limite u a)
  (hv : limite v c)
  (hle : ∀ n, u n ≤ v n)
  : a ≤ c :=
begin
  apply le_of_not_lt,
  intro hlt,
  set ε := (a - c) /2 with hε,
  cases hu ε (by linarith) with Nu HNu,
  cases hv ε (by linarith) with Nv HNv,
  let N := max Nu Nv,
  have Ha : |u N - a| < ε :=
    HNu N (le_max_left Nu Nv),
  have Hc : |v N - c| < ε :=
    HNv N (le_max_right Nu Nv),
  have HN : u N ≤ v N := hle N,
  apply lt_irrefl (a - c),
  calc a - c
       = (a - u N) + (u N - c)    : by ring
   ... ≤ (a - u N) + (v N - c)    : by simp [HN]
   ... ≤ |(a - u N) + (v N - c)| : le_abs_self ((a - u N) + (v N - c))
   ... ≤ |a - u N| + |v N - c|    : abs_add (a - u N) (v N - c)
   ... = |u N - a| + |v N - c|    : by simp only [abs_sub]
   ... < ε + ε                    : add_lt_add Ha Hc
   ... = a - c                    : add_halves (a - c),
end

-- 3ª demostración
```

```
example
  (hu : limite u a)
  (hv : limite v c)
  (hle : ∀ n, u n ≤ v n)
  : a ≤ c :=
begin
  apply le_of_not_lt,
  intro hlt,
  set ε := (a - c) /2 with hε,
  cases hu ε (by linarith) with Nu HNu,
  cases hv ε (by linarith) with Nv HNv,
  let N := max Nu Nv,
  have Ha : |u N - a| < ε :=
    HNu N (le_max_left Nu Nv),
  have Hc : |v N - c| < ε :=
    HNv N (le_max_right Nu Nv),
  have HN : u N ≤ v N := hle N,
  apply lt_irrefl (a - c),
  calc a - c
       = (a - u N) + (u N - c)   : by ring
   ... ≤ (a - u N) + (v N - c)   : by simp [HN]
   ... ≤ |(a - u N) + (v N - c)| : by simp [le_abs_self]
   ... ≤ |a - u N| + |v N - c|   : by simp [abs_add]
   ... = |u N - a| + |v N - c|   : by simp [abs_sub]
   ... < ε + ε                   : add_lt_add Ha Hc
   ... = a - c                   : by simp,
end

-- 4ª demostración
example
  (hu : limite u a)
  (hv : limite v c)
  (hle : ∀ n, u n ≤ v n)
  : a ≤ c :=
begin
  apply le_of_not_lt,
  intro hlt,
  set ε := (a - c) /2 with hε,
  cases hu ε (by linarith) with Nu HNu,
  cases hv ε (by linarith) with Nv HNv,
  let N := max Nu Nv,
  have Ha : |u N - a| < ε :=
    HNu N (le_max_left Nu Nv),
  have Hc : |v N - c| < ε :=
    HNv N (le_max_right Nu Nv),
```

```
  have HN : u N ≤ v N := hle N,
  apply lt_irrefl (a - c),
  rw abs_lt at Ha Hc,
  linarith,
end
```

Se puede interactuar con las pruebas anteriores en esta sesión con Lean.

4.30. Las sucesiones acotadas por cero son nulas

```
-- ---------------------------------------------------------------------
-- Demostrar que las sucesiones acotadas por cero son nulas.
-- ---------------------------------------------------------------------

import data.real.basic
import tactic

variable (u : ℕ → ℝ)

notation `|`x`|` := abs x

-- 1ª demostración
example
  (h : ∀ n, |u n| ≤ 0)
  : ∀ n, u n = 0 :=
begin
  intro n,
  rw ← abs_eq_zero,
  specialize h n,
  apply le_antisymm,
  { exact h, },
  { exact abs_nonneg (u n), },
end

-- 2ª demostración
example
  (h : ∀ n, |u n| ≤ 0)
  : ∀ n, u n = 0 :=
begin
  intro n,
  rw ← abs_eq_zero,
```

```
  specialize h n,
  exact le_antisymm h (abs_nonneg (u n)),
end

-- 3ª demostración
example
  (h : ∀ n, |u n| ≤ 0)
  : ∀ n, u n = 0 :=
begin
  intro n,
  rw ← abs_eq_zero,
  exact le_antisymm (h n) (abs_nonneg (u n)),
end

-- 4ª demostración
example
  (h : ∀ n, |u n| ≤ 0)
  : ∀ n, u n = 0 :=
begin
  intro n,
  exact abs_eq_zero.mp (le_antisymm (h n) (abs_nonneg (u n))),
end

-- 5ª demostración
example
  (h : ∀ n, |u n| ≤ 0)
  : ∀ n, u n = 0 :=
λ n, abs_eq_zero.mp (le_antisymm (h n) (abs_nonneg (u n)))

-- 6ª demostración
example
  (h : ∀ n, |u n| ≤ 0)
  : ∀ n, u n = 0 :=
by finish
```

Se puede interactuar con las pruebas anteriores en esta sesión con Lean.

4.31. Producto de una sucesión acotada por otra convergente a cero

```
-- ---------------------------------------------------------------------
-- Demostrar que el producto de una sucesión acotada por una convergente
-- a 0 también converge a 0.
-- ---------------------------------------------------------------------

import data.real.basic
import tactic

variables (u v : ℕ → ℝ)
variable  (a : ℝ)

notation `|`x`|` := abs x

def limite (u : ℕ → ℝ) (c : ℝ) :=
∀ ε > 0, ∃ N, ∀ n ≥ N, |u n - c| < ε

def acotada (a : ℕ → ℝ) :=
∃ B, ∀ n, |a n| ≤ B

-- 1ª demostración
example
  (hU : acotada u)
  (hV : limite v 0)
  : limite (u*v) 0 :=
begin
  cases hU with B hB,
  have hBnoneg : 0 ≤ B,
    calc 0 ≤ |u 0| : abs_nonneg (u 0)
       ... ≤ B     : hB 0,
  by_cases hB0 : B = 0,
  { subst hB0,
    intros ε hε,
    use 0,
    intros n hn,
    simp_rw [sub_zero] at *,
    calc |(u * v) n|
         = |u n * v n|    : congr_arg abs (pi.mul_apply u v n)
     ... = |u n| * |v n| : abs_mul (u n) (v n)
     ... ≤ 0 * |v n|     : mul_le_mul_of_nonneg_right (hB n) (abs_nonneg (v n))
     ... = 0              : zero_mul (|v n|)
     ... < ε              : hε, },
  { change B ≠ 0 at hB0,
    have hBpos : 0 < B := (ne.le_iff_lt hB0.symm).mp hBnoneg,
    intros ε hε,
    cases hV (ε/B) (div_pos hε hBpos) with N hN,
```

```
    use N,
    intros n hn,
    simp_rw [sub_zero] at *,
    calc |(u * v) n|
         = |u n * v n|     : congr_arg abs (pi.mul_apply u v n)
     ... = |u n| * |v n|   : abs_mul (u n) (v n)
     ... ≤ B * |v n|       : mul_le_mul_of_nonneg_right (hB n) (abs_nonneg _)
     ... < B * (ε/B)       : mul_lt_mul_of_pos_left (hN n hn) hBpos
     ... = ε               : mul_div_cancel' ε hB0 },
end

-- 2ª demostración
example
  (hU : acotada u)
  (hV : limite v 0)
  : limite (u*v) 0 :=
begin
  cases hU with B hB,
  have hBnoneg : 0 ≤ B,
    calc 0 ≤ |u 0| : abs_nonneg (u 0)
       ... ≤ B     : hB 0,
  by_cases hB0 : B = 0,
  { subst hB0,
    intros ε hε,
    use 0,
    intros n hn,
    simp_rw [sub_zero] at *,
    calc |(u * v) n|
         = |u n| * |v n| : by finish [abs_mul]
     ... ≤ 0 * |v n|     : mul_le_mul_of_nonneg_right (hB n) (abs_nonneg (v n))
     ... = 0             : by ring
     ... < ε             : hε, },
  { change B ≠ 0 at hB0,
    have hBpos : 0 < B := (ne.le_iff_lt hB0.symm).mp hBnoneg,
    intros ε hε,
    cases hV (ε/B) (div_pos hε hBpos) with N hN,
    use N,
    intros n hn,
    simp_rw [sub_zero] at *,
    calc |(u * v) n|
         = |u n| * |v n|   : by finish [abs_mul]
     ... ≤ B * |v n|       : mul_le_mul_of_nonneg_right (hB n) (abs_nonneg _)
     ... < B * (ε/B)       : by finish
     ... = ε               : mul_div_cancel' ε hB0 },
end
```

Se puede interactuar con las pruebas anteriores en esta sesión con Lean.

Capítulo 5

Ejercicios de agosto de 2021

5.1. La congruencia módulo 2 es una relación de equivalencia

```
-- ----------------------------------------------------------------------
-- Se define la relación R entre los números enteros de forma que x está
-- relacionado con y si x-y es divisible por 2. Demostrar que R es una
-- relación de equivalencia.
-- ----------------------------------------------------------------------

import data.int.basic
import tactic

def R (m n : ℤ) := 2 ∣ (m - n)

-- 1ª demostración
example : equivalence R :=
begin
  repeat {split},
  { intro x,
    unfold R,
    rw sub_self,
    exact dvd_zero 2, },
  { intros x y hxy,
    unfold R,
    cases hxy with a ha,
    use -a,
    calc y - x
         = -(x - y) : (neg_sub x y).symm
     ... = -(2 * a) : by rw ha
```

```
     ... = 2 * -a   : neg_mul_eq_mul_neg 2 a, },
  { intros x y z hxy hyz,
    cases hxy with a ha,
    cases hyz with b hb,
    use a + b,
    calc x - z
         = (x - y) + (y - z) : (sub_add_sub_cancel x y z).symm
     ... = 2 * a + 2 * b     : congr_arg2 ((+)) ha hb
     ... = 2 * (a + b)       : (mul_add 2 a b).symm , },
end

-- 2ª demostración
example : equivalence R :=
begin
  repeat {split},
  { intro x,
    simp [R], },
  { rintros x y ⟨a, ha⟩,
    use -a,
    linarith, },
  { rintros x y z ⟨a, ha⟩ ⟨b, hb⟩,
    use a + b,
    linarith, },
end
```

Se puede interactuar con las pruebas anteriores en esta sesión con Lean.

5.2. Las funciones con inversa por la izquierda son inyectivas

```
-- ---------------------------------------------------------------------
-- En Lean, que g es una inversa por la izquierda de f está definido por
--    left_inverse (g : β → α) (f : α → β) : Prop :=
--       ∀ x, g (f x) = x
-- y que f tenga inversa por la izquierda está definido por
--    has_left_inverse (f : α → β) : Prop :=
--       ∃ finv : β → α, left_inverse finv f
-- Finalmente, que f es inyectiva está definido por
--    injective (f : α → β) : Prop :=
--       ∀ ⦃x y⦄, f x = f y → x = y
--
-- Demostrar que si f tiene inversa por la izquierda, entonces f es
```

```
-- inyectiva.
-- ----------------------------------------------------------------------

import tactic
open function

universes u v
variables {α : Type u}
variable  {β : Type v}
variable  {f : α → β}

-- 1ª demostración
example
  (hf : has_left_inverse f)
  : injective f :=
begin
  intros x y hxy,
  unfold has_left_inverse at hf,
  unfold left_inverse at hf,
  cases hf with g hg,
  calc x = g (f x) : (hg x).symm
     ... = g (f y) : congr_arg g hxy
     ... = y       : hg y
end

-- 2ª demostración
example
  (hf : has_left_inverse f)
  : injective f :=
begin
  intros x y hxy,
  cases hf with g hg,
  calc x = g (f x) : (hg x).symm
     ... = g (f y) : congr_arg g hxy
     ... = y       : hg y
end

-- 3ª demostración
example
  (hf : has_left_inverse f)
  : injective f :=
exists.elim hf (λ finv inv, inv.injective)

-- 4ª demostración
example
```

```
  (hf : has_left_inverse f)
  : injective f :=
has_left_inverse.injective hf
```

Se puede interactuar con las pruebas anteriores en esta sesión con Lean.

5.3. Las funciones inyectivas tienen inversa por la izquierda

```
-- ---------------------------------------------------------------------
-- En Lean, que g es una inversa por la izquierda de f está definido por
--    left_inverse (g : β → α) (f : α → β) : Prop :=
--       ∀ x, g (f x) = x
-- y que f tenga inversa por la izquierda está definido por
--    has_left_inverse (f : α → β) : Prop :=
--       ∃ finv : β → α, left_inverse finv f
-- Finalmente, que f es inyectiva está definido por
--    injective (f : α → β) : Prop :=
--       ∀ ⦃x y⦄, f x = f y → x = y
--
-- Demostrar que si f es una función inyectiva con dominio no vacío,
-- entonces f tiene inversa por la izquierda.
-- ---------------------------------------------------------------------

import tactic
open function classical

variables {α β: Type*}
variable  {f : α → β}

-- 1ª demostración
example
  [hα : nonempty α]
  (hf : injective f)
  : has_left_inverse f :=
begin
  classical,
  unfold has_left_inverse,
  let g := λ y, if h : ∃ x, f x = y then some h else choice hα,
  use g,
  unfold left_inverse,
  intro a,
```

```
  have h1 : ∃ x : α, f x = f a := Exists.intro a rfl,
  dsimp at *,
  dsimp [g],
  rw dif_pos h1,
  apply hf,
  exact some_spec h1,
end

-- 2ª demostración
example
  [hα : nonempty α]
  (hf : injective f)
  : has_left_inverse f :=
begin
  classical,
  let g := λ y, if h : ∃ x, f x = y then some h else choice hα,
  use g,
  intro a,
  have h1 : ∃ x : α, f x = f a := Exists.intro a rfl,
  dsimp [g],
  rw dif_pos h1,
  exact hf (some_spec h1),
end

-- 3ª demostración
example
  [hα : nonempty α]
  (hf : injective f)
  : has_left_inverse f :=
begin
  unfold has_left_inverse,
  use inv_fun f,
  unfold left_inverse,
  intro x,
  apply hf,
  apply inv_fun_eq,
  use x,
end

-- 4ª demostración
example
  [hα : nonempty α]
  (hf : injective f)
  : has_left_inverse f :=
begin
```

```
  use inv_fun f,
  intro x,
  apply hf,
  apply inv_fun_eq,
  use x,
end

-- 5ª demostración
example
  [hα : nonempty α]
  (hf : injective f)
  : has_left_inverse f :=
⟨inv_fun f, left_inverse_inv_fun hf⟩

-- 6ª demostración
example
  [hα : nonempty α]
  (hf : injective f)
  : has_left_inverse f :=
injective.has_left_inverse hf
```

Se puede interactuar con las pruebas anteriores en esta sesión con Lean.

5.4. Una función tiene inversa por la izquierda si y solo si es inyectiva

```
-- ---------------------------------------------------------------------
-- En Lean, que g es una inversa por la izquierda de f está definido por
--    left_inverse (g : β → α) (f : α → β) : Prop :=
--       ∀ x, g (f x) = x
-- y que f tenga inversa por la izquierda está definido por
--    has_left_inverse (f : α → β) : Prop :=
--       ∃ finv : β → α, left_inverse finv f
-- Finalmente, que f es inyectiva está definido por
--    injective (f : α → β) : Prop :=
--       ∀ ⦃x y⦄, f x = f y → x = y
--
-- Demostrar que una función f, con dominio no vacío, tiene inversa por
-- la izquierda si y solo si es inyectiva.
-- ---------------------------------------------------------------------

import tactic
```

```
open function

variables {α : Type*} [nonempty α]
variable  {β : Type*}
variable  {f : α → β}

-- 1ª demostración
example : has_left_inverse f ↔ injective f :=
begin
  split,
  { intro hf,
    intros x y hxy,
    cases hf with g hg,
    calc x = g (f x) : (hg x).symm
       ... = g (f y) : congr_arg g hxy
       ... = y       : hg y, },
  { intro hf,
    use inv_fun f,
    intro x,
    apply hf,
    apply inv_fun_eq,
    use x, },
end

-- 2ª demostración
example : has_left_inverse f ↔ injective f :=
begin
  split,
  { intro hf,
    exact has_left_inverse.injective hf },
  { intro hf,
    exact injective.has_left_inverse hf },
end

-- 3ª demostración
example : has_left_inverse f ↔ injective f :=
⟨has_left_inverse.injective, injective.has_left_inverse⟩

-- 4ª demostración
example : has_left_inverse f ↔ injective f :=
injective_iff_has_left_inverse.symm
```

Se puede interactuar con las pruebas anteriores en esta sesión con Lean.

5.5. Las funciones con inversa por la derecha son suprayectivas

```
-- ---------------------------------------------------------------------
-- En Lean, que g es una inversa por la izquierda de f está definido por
--    left_inverse (g : β → α) (f : α → β) : Prop :=
--       ∀ x, g (f x) = x
-- que g es una inversa por la derecha de f está definido por
--    right_inverse (g : β → α) (f : α → β) : Prop :=
--       left_inverse f g
-- y que f tenga inversa por la derecha está definido por
--    has_right_inverse (f : α → β) : Prop :=
--       ∃ g : β → α, right_inverse g f
-- Finalmente, que f es suprayectiva está definido por
--    def surjective (f : α → β) : Prop :=
--       ∀ b, ∃ a, f a = b
--
-- Demostrar que si la función f tiene inversa por la derecha, entonces
-- f es suprayectiva.
-- ---------------------------------------------------------------------

import tactic
open function

variables {α β: Type*}
variable  {f : α → β}

-- 1ª demostración
example
  (hf : has_right_inverse f)
  : surjective f :=
begin
  unfold surjective,
  unfold has_right_inverse at hf,
  cases hf with g hg,
  intro b,
  use g b,
  exact hg b,
end

-- 2ª demostración
example
  (hf : has_right_inverse f)
  : surjective f :=
```

```
begin
  intro b,
  cases hf with g hg,
  use g b,
  exact hg b,
end

-- 3ª demostración
example
  (hf : has_right_inverse f)
  : surjective f :=
begin
  intro b,
  cases hf with g hg,
  use [g b, hg b],
end

-- 4ª demostración
example
  (hf : has_right_inverse f)
  : surjective f :=
has_right_inverse.surjective hf
```

Se puede interactuar con las pruebas anteriores en esta sesión con Lean.

5.6. Las funciones suprayectivas tienen inversa por la derecha

```
-- ---------------------------------------------------------------------
-- En Lean, que g es una inversa por la izquierda de f está definido por
--    left_inverse (g : β → α) (f : α → β) : Prop :=
--       ∀ x, g (f x) = x
-- que g es una inversa por la derecha de f está definido por
--    right_inverse (g : β → α) (f : α → β) : Prop :=
--       left_inverse f g
-- y que f tenga inversa por la derecha está definido por
--    has_right_inverse (f : α → β) : Prop :=
--       ∃ g : β → α, right_inverse g f
-- Finalmente, que f es suprayectiva está definido por
--    def surjective (f : α → β) : Prop :=
--       ∀ b, ∃ a, f a = b
--
```

```
-- Demostrar que si f es una función suprayectiva, entonces f tiene
-- inversa por la derecha.
-- ----------------------------------------------------------------------

import tactic
open function classical

variables {α β: Type*}
variable  {f : α → β}

-- 1ª demostración
example
  (hf : surjective f)
  : has_right_inverse f :=
begin
  unfold has_right_inverse,
  let g := λ y, some (hf y),
  use g,
  unfold function.right_inverse,
  unfold function.left_inverse,
  intro b,
  apply some_spec (hf b),
end

-- 2ª demostración
example
  (hf : surjective f)
  : has_right_inverse f :=
begin
  let g := λ y, some (hf y),
  use g,
  intro b,
  apply some_spec (hf b),
end

-- 3ª demostración
example
  (hf : surjective f)
  : has_right_inverse f :=
begin
  use surj_inv hf,
  intro b,
  exact surj_inv_eq hf b,
end
```

```
-- 4ª demostración
example
  (hf : surjective f)
  : has_right_inverse f :=
begin
  use surj_inv hf,
  exact surj_inv_eq hf,
end

-- 5ª demostración
example
  (hf : surjective f)
  : has_right_inverse f :=
begin
  use [surj_inv hf, surj_inv_eq hf],
end

-- 6ª demostración
example
  (hf : surjective f)
  : has_right_inverse f :=
⟨surj_inv hf, surj_inv_eq hf⟩

-- 7ª demostración
example
  (hf : surjective f)
  : has_right_inverse f :=
⟨_, surj_inv_eq hf⟩

-- 8ª demostración
example
  (hf : surjective f)
  : has_right_inverse f :=
surjective.has_right_inverse hf
```

Se puede interactuar con las pruebas anteriores en esta sesión con Lean.

5.7. Una función tiene inversa por la derecha si y solo si es suprayectiva

```
-- ---------------------------------------------------------------------
-- En Lean, que g es una inversa por la izquierda de f está definido por
--    left_inverse (g : β → α) (f : α → β) : Prop :=
--       ∀ x, g (f x) = x
-- que g es una inversa por la derecha de f está definido por
--    right_inverse (g : β → α) (f : α → β) : Prop :=
--       left_inverse f g
-- y que f tenga inversa por la derecha está definido por
--    has_right_inverse (f : α → β) : Prop :=
--       ∃ g : β → α, right_inverse g f
-- Finalmente, que f es suprayectiva está definido por
--    def surjective (f : α → β) : Prop :=
--       ∀ b, ∃ a, f a = b
--
-- Demostrar que la función f tiene inversa por la derecha si y solo si
-- es suprayectiva.
-- ---------------------------------------------------------------------

import tactic
open function classical

variables {α β: Type*}
variable  {f : α → β}

-- 1ª demostración
example : has_right_inverse f ↔ surjective f :=
begin
  split,
  { intros hf b,
    cases hf with g hg,
    use g b,
    exact hg b, },
  { intro hf,
    let g := λ y, some (hf y),
    use g,
    intro b,
    apply some_spec (hf b), },
end

-- 2ª demostración
example : has_right_inverse f ↔ surjective f :=
surjective_iff_has_right_inverse.symm
```

Se puede interactuar con las pruebas anteriores en esta sesión con Lean.

5.8. Las funciones con inversa son biyectivas

```
-- ---------------------------------------------------------------------
-- En Lean se puede definir que g es una inversa de f por
--    def inversa (f : X → Y) (g : Y → X) :=
--      (∀ x, (g ∘ f) x = x) ∧ (∀ y, (f ∘ g) y = y)
-- y que f tiene inversa por
--    def tiene_inversa (f : X → Y) :=
--      ∃ g, inversa g f
--
-- Demostrar que si la función f tiene inversa, entonces f es biyectiva.
-- ---------------------------------------------------------------------

import tactic
open function

variables {X Y : Type*}
variable  (f : X → Y)

def inversa (f : X → Y) (g : Y → X) :=
  (∀ x, (g ∘ f) x = x) ∧ (∀ y, (f ∘ g) y = y)

def tiene_inversa (f : X → Y) :=
  ∃ g, inversa g f

-- 1ª demostración
example
  (hf : tiene_inversa f)
  : bijective f :=
begin
  rcases hf with ⟨g, ⟨h1, h2⟩⟩,
  split,
  { intros a b hab,
    calc a = g (f a) : (h2 a).symm
       ... = g (f b) : congr_arg g hab
       ... = b       : h2 b, },
  { intro y,
    use g y,
    exact h1 y, },
end

-- 2ª demostración
example
  (hf : tiene_inversa f)
```

```
  : bijective f :=
begin
  rcases hf with ⟨g, ⟨h1, h2⟩⟩,
  split,
  { intros a b hab,
    calc a = g (f a) : (h2 a).symm
       ... = g (f b) : congr_arg g hab
       ... = b       : h2 b, },
  { intro y,
    use [g y, h1 y], },
end

-- 3ª demostración
example
  (hf : tiene_inversa f)
  : bijective f :=
begin
  rcases hf with ⟨g, ⟨h1, h2⟩⟩,
  split,
  { exact left_inverse.injective h2, },
  { exact right_inverse.surjective h1, },
end

-- 4ª demostración
example
  (hf : tiene_inversa f)
  : bijective f :=
begin
  rcases hf with ⟨g, ⟨h1, h2⟩⟩,
  exact ⟨left_inverse.injective h2,
         right_inverse.surjective h1⟩,
end

-- 5ª demostración
example :
  tiene_inversa f → bijective f :=
begin
  rintros ⟨g, ⟨h1, h2⟩⟩,
  exact ⟨left_inverse.injective h2,
         right_inverse.surjective h1⟩,
end

-- 6ª demostración
example :
  tiene_inversa f → bijective f :=
```

```
λ ⟨g, ⟨h1, h2⟩⟩, ⟨left_inverse.injective h2,
                  right_inverse.surjective h1⟩
```

Se puede interactuar con las pruebas anteriores en esta sesión con Lean.

Bibliografía

[1] J. A. Alonso. Lógica con Lean [1], 2020.

[2] J. A. Alonso. Lean para matemáticos [2], 2021.

[3] J. A. Alonso. Matemáticas en Lean [3], 2021.

[4] J. A. Alonso. DAO (Demostración Asistida por Ordenador) con Lean [4], 2021.

[5] J. Avigad, K. Buzzard, R. Y. Lewis, and P. Massot. Mathematics in Lean [5], 2020.

[6] J. Avigad, L. de Moura, and S. Kong. Theorem Proving in Lean [6], 2021.

[7] J. Avigad, R. Y. Lewis, and F. van Doorn. Logic and proof [7], 2020.

[8] A. Baanen, A. Bentkamp, J. Blanchette, J. Hölzl, and J. Limperg. The Hitchhiker's Guide to Logical Verification [8], 2020.

[9] K. Buzzard. Course on formalising mathematics [9], 2021.

[10] K. Buzzard and M. Pedramfar. The Natural Number Game, version 1.3.3 [10].

---

[1] https://raw.githubusercontent.com/jaalonso/Logica_con_Lean/master/Logica_con_Lean.pdf

[2] https://github.com/jaalonso/Lean_para_matematicos

[3] https://github.com/jaalonso/Matematicas_en_Lean

[4] https://raw.githubusercontent.com/jaalonso/DAO_con_Lean/master/DAO_con_Lean.pdf

[5] https://leanprover-community.github.io/mathematics_in_lean/

[6] https://leanprover.github.io/theorem_proving_in_lean/theorem_proving_in_lean.pdf

[7] https://leanprover.github.io/logic_and_proof

[8] https://raw.githubusercontent.com/blanchette/logical_verification_2020/master/hitchhikers_guide.pdf

[9] https://github.com/ImperialCollegeLondon/formalising-mathematics

[10] https://www.ma.imperial.ac.uk/~buzzard/xena/natural_number_game/

[11] P. Massot. Introduction aux mathématiques formalisées [11].

[12] Varios. LFTCM 2020: Lean for the Curious Mathematician 2020 [12].

[11] https://www.imo.universite-paris-saclay.fr/~pmassot/enseignement/math114/

[12] https://leanprover-community.github.io/lftcm2020/schedule.html