Matemáticas en Lean4

José A. Alonso Jiménez

Grupo de Lógica Computacional
Dpto. de Ciencias de la Computación e Inteligencia Artificial
Universidad de Sevilla
Sevilla, 14 de julio de 2023

Índice general

Capítulo 1

Introducción

1.1. Resumen

El objetivo de este trabajo es presentar el uso de Lean4 (y su librería matemática mathlib4) mediante ejemplos matemáticos. Está basado en el libro Mathematics in Lean de Jeremy Avigad y Patrick Massot.

Los ejercicios se han ido publicando, desde el 10 de julio de 2022, en el blog Calculemus y su código en GitHub.

#+BEGIN_COMMENT

1.2. Creación del proyecto

- Se crea con

```
lake +leanprover/lean4:nightly-2023-07-06 new Matematicas_en_Lean4 math
cd Matematicas_en_Lean4/
lake update
lake exe cache get
```

#+END_COMMENT

1.3. Presentación panorámica de Lean

1.3.1. Ejemplo de evaluación

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

#eval 2 + 3

-- Comentario: Al poner el cursor sobre eval se escribe su resultado al
-- final de la línea.
```

1.3.2. Ejemplo de comprobación con check

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

#check 2 + 3

-- Comentario: Al colocar el cursor sobre check escribe al final de la
-- línea
--    2 + 3 : Nat
-- que indica que el valor de la expresión es un número natural.
```

1.3.3. Ejemplo de definición de funciones

```
-- ---------------------------------------------------------------------
-- Ejercicio. Importar la teoría de los números naturales.
-- ---------------------------------------------------------------------

import Mathlib.Data.Nat.Basic

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

def f (x : ℕ) :=
  x + 3

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

```
#check f

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

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

#eval f 2

-- Comentario: Al colocar el cursor sobre eval escribe su valor (5).
```

1.3.4. Ejemplo de proposiciones

```
-- ---------------------------------------------------------------------
-- Ejercicio. Importar la teoría de los números naturales.
-- ---------------------------------------------------------------------

import Mathlib.Data.Nat.Basic

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

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

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

#check ultimo_teorema_de_Fermat

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

1.3.5. Ejemplo de teoremas

```
-- ---------------------------------------------------------------------
-- Ejercicio. Importar la teoría de los números naturales.
-- ---------------------------------------------------------------------

import Mathlib.Data.Nat.Basic

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

theorem facil : 2 + 3 = 5 := rfl

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

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

#check facil

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

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

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

theorem dificil : ultimo_teorema_de_Fermat :=
sorry

-- Comentarios:
-- 1. La palabra sorry se usa para omitir la demostración.
-- 2. Se puede verificar la teoría pulsando
--       C-c ! l
```

```
--    Se obtiene
--       Line Col Level     Message
--    24   1 info      facil : 2 + 3 = 5 (lsp)
--    37   9 warning   declaration uses 'sorry' (lsp)

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

#check dificil

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

1.3.6. Ejemplo de demostración

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

import Mathlib.Data.Nat.Basic
import Mathlib.Data.Nat.Parity
import Mathlib.Tactic

open Nat

-- 1ª demostración
example : ∀ m n : ℕ, Even n → Even (m * n) := by
  rintro m n ⟨k, hk⟩
  use m * k
  rw [hk]
  ring

-- 2ª demostración
example : ∀ m n : ℕ, Even n → Even (m * n) := by
  rintro m n ⟨k, hk⟩
  use m * k
  rw [hk]
  rw [mul_add]

-- 3ª demostración
example : ∀ m n : ℕ, Even n → Even (m * n) := by
```

```
  rintro m n ⟨k, hk⟩
  use m * k
  rw [hk, mul_add]

-- 4ª demostración
example : ∀ m n : Nat, Even n → Even (m * n) := by
  rintro m n ⟨k, hk⟩; use m * k; rw [hk, mul_add]

-- 5ª demostración
example : ∀ m n : ℕ, Even n → Even (m * n) := by
  rintro m n ⟨k, hk⟩
  exact ⟨m * k, by rw [hk, mul_add]⟩

-- 6ª demostración
example : ∀ m n : Nat, Even n → Even (m * n) :=
fun m n ⟨k, hk⟩ ↦ ⟨m * k, by rw [hk, mul_add]⟩

-- 7ª demostración
example : ∀ m n : ℕ, Even n → Even (m * n) := by
  rintro m n ⟨k, hk⟩
  use m * k
  rw [hk]
  exact mul_add m k k

-- 8ª demostración
example : ∀ m n : ℕ, Even n → Even (m * n) := by
  intros m n hn
  unfold Even at *
  cases hn with
  | intro k hk =>
    use m * k
    rw [hk, mul_add]

-- 9ª demostración
example : ∀ m n : ℕ, Even n → Even (m * n) := by
  intros m n hn
  unfold Even at *
  cases hn with
  | intro k hk =>
    use m * k
    calc m * n
       = m * (k + k)   := by exact congrArg (HMul.hMul m) hk
     _ = m * k + m * k := by exact mul_add m k k

-- 10ª demostración
```

```
example : ∀ m n : Nat, Even n → Even (m * n) := by
  intros; simp [*, parity_simps]

-- Comentarios:
-- 1. Al poner el curso en la línea 1 sobre Mathlib.Data.Nat.Parity y pulsar M-.
--    se abre la teoría correspondiente.
-- 2. Al colocar el cursor sobre el nombre de un lema se ve su enunciado.
-- 3. Para completar el nombre de un lema basta escribir parte de su
--    nombre y completar con S-SPC (es decir, simultáneamente las teclas
--    de mayúscula y la de espacio).
-- 4. El lema que se ha usado es
--       mul_add a b c : a * (b + c) = a * b + a * c
-- 4. Se activa la ventana de objetivos (*Lean Goal*) pulsando C-c TAB
-- 5. Al mover el cursor sobre las pruebas se actualiza la ventana de
--    objetivos.
```

Demostraciones comentadas

Las demostraciones presentadas tienen como objetivo demostrar la proposición de que, para cualquier número natural `m` y `n`, si `n` es par (`Even`), entonces el producto de `m` y `n` también es par. A continuación, analizaremos cada demostración en detalle:

La **1ª demostración** comienza con el comando `example`, que establece la meta que se desea demostrar. Luego, se introducen las variables `m` y `n` utilizando el comando `rintro`, lo que permite utilizarlas en la prueba. A continuación, se introduce la hipótesis de que `n` es par utilizando `⟨k, hk⟩`, donde `k` es un número natural y `hk` es una prueba de que `n = k + k`.

Para demostrar que `m * n` es par, se utiliza el comando `use m * k`, que establece `m * k` como el número natural que demostrará que `m * n` es par. Luego, se utiliza el comando `rw [hk]` para reemplazar `n` en la meta con `k + k`, utilizando la prueba `hk`.

Finalmente, se utiliza el comando `ring` para simplificar la expresión `m * (k + k)` a `m * k + m * k` utilizando las propiedades algebraicas de los números naturales.

En resumen, la demostración establece que si `n` es par, entonces `m * n` también es par, utilizando la propiedad de la paridad de los números naturales.

La **2ª demostración** es similar a la primera, pero incluye un paso adicional utilizando el comando `rw [mul_add]`.

Al igual que en la primera demostración, se comienza con el comando `example`, se introducen las variables `m` y `n` con `rintro`, y se establece la hipó-

tesis de que `n` es par utilizando `⟨k, hk⟩`. A continuación, se utiliza el comando `use m * k` para establecer `m * k` como el número natural que demostrará que `m * n` es par.

Después de eso, se utiliza el comando `rw [hk]` para reemplazar `n` en la meta por `k *+ k`, utilizando la prueba `hk`. Esto es similar a la primera demostración.

Sin embargo, en la segunda demostración se agrega el comando `rw [mul_add]` adicionalmente. Este comando utiliza la propiedad distributiva de la multiplicación respecto a la adición en los números naturales. Al aplicar `rw [mul_add]`, se expande la expresión `m * (k + k)` en la meta a `m * k + m * k`.

En resumen, la segunda demostración es una extensión de la primera, donde se agrega el paso adicional `rw [mul_add]` para aplicar la propiedad distributiva de la multiplicación respecto a la adición en los números naturales. Esto permite expandir la expresión `m * (k + k)` a `m * k + m * k` en la meta de la demostración.

La **3ª demostración** comienza con el comando `rintro` seguido de `m n ⟨k, hk⟩`. Esto introduce las variables `m` y `n`, así como una hipótesis `hk` de la forma `⟨k, hk⟩` que establece que `n` es par.

Luego se utiliza `use m * k` para establecer `m * k` como el número demostrará que `m * n` es par. Esto se logra al utilizar el comando `rw` (reemplazo) con `[hk, mul_add]`. El reemplazo se realiza en dos pasos. Primero, se reemplaza `hk`, lo que resulta en la sustitución de `n` por `k + k` en la meta. Luego, se aplica la propiedad distributiva de la multiplicación respecto a la adición en los números naturales, expandiendo la expresión `m * (k + k)` a `m * k + m * k`.

En resumen, la tercera demostración utiliza la introducción de variables, la asignación de un número específico (`m * k`) para demostrar la paridad de `m * n`, y luego realiza reemplazos para simplificar la expresión y llegar al resultado deseado.

La **4ª demostración** utiliza una notación más compacta en comparación con las demostraciones anteriores. Comienza con el comando `rintro m n ⟨k, hk⟩`, que introduce las variables `m` y `n`, así como una hipótesis `hk` de la forma `⟨k, hk⟩` que establece que `n` es par.

Luego se utiliza `use m * k` para establecer `m * k` como el número que demostrará que `m * n` es par. Esta es una abreviatura de `use (m * k)`.

A continuación, se utiliza el comando `rw [hk, mul_add]` para realizar dos reemplazos en un solo paso. Primero, se reemplaza `hk`, lo que sustituye `n` por `k + k` en la meta. Luego, se aplica la propiedad distributiva de la multiplicación respecto a la adición en los números naturales, expandiendo la expresión `m * (k + k)` a `m * k + m * k`.

En resumen, la cuarta demostración utiliza una notación más compacta

para introducir variables y establecer el número que demuestra la paridad de `m * n`, y luego realiza los reemplazos necesarios para simplificar la expresión y alcanzar el resultado deseado.

La **5ª demostración** comienza con el comando `rintro` para introducir las variables `m` y `n`, y luego `⟨k, hk⟩` se utiliza para establecer la hipótesis de paridad `n = k + k`, donde `k` es un número natural y `hk` es una prueba de esta igualdad.

Luego se utiliza el comando `exact ⟨m * k, by rw [hk, mul_add]⟩` para establecer directamente `⟨m * k, …⟩` como la prueba requerida de que `m * n` es par. Aquí, `⟨m * k, …⟩` representa el número `m * k` como testigo de la paridad, y `by rw [hk, mul_add]` proporciona una prueba que muestra que ese número es par.

Dentro de `by rw [hk, mul_add]`, se realiza el reemplazo utilizando `rw`. Primero, se reemplaza `hk`, lo que sustituye `n` por `k + k` en la meta. Luego, se aplica la propiedad distributiva de la multiplicación respecto a la adición en los números naturales, expandiendo la expresión `m * (k + k)` a `m * k + m * k`.

En resumen, la quinta demostración utiliza el comando `rintro` para introducir las variables y la hipótesis de paridad, y luego utiliza `exact` para establecer directamente el número y la prueba requeridos para demostrar la paridad de `m * n`. Proporciona una solución directa y concisa al problema planteado.

La **6ª demostración** utiliza una notación de función lambda para definir directamente la prueba requerida. Comienza con `fun m n ⟨k, hk⟩ ↦`, donde se introducen las variables `m` y `n`, y se establece una hipótesis `⟨k, hk⟩` que afirma que `n` es par.

Luego, se utiliza `⟨m * k, by rw [hk, mul_add]⟩` para establecer directamente `m * k` como el número que demostrará que `m * n` es par. Esto se hace mediante el uso de la notación `⟨valor, prueba⟩`, donde `valor` representa el número que se utilizará como testigo de la paridad y `prueba` es una prueba que muestra que ese valor es par.

En este caso, se establece `m * k` como el valor y se proporciona prueba utilizando by `rw [hk, mul_add]`. Aquí, `rw [hk, mul_add]` realiza dos reemplazos en un solo paso. Primero, se reemplaza `hk`, lo que sustituye `n` por `k + k` en la meta. Luego, se aplica la propiedad distributiva de la multiplicación respecto a la adición en los números naturales, expandiendo la expresión `m * (k + k)` a `m * k + m * k`.

En resumen, la sexta demostración utiliza una función lambda para definir directamente el valor y la prueba necesarios para demostrar la paridad de `m * n`. Proporciona una solución concisa y directa al problema planteado, al igual que la quinta demostración.

La **7ª demostración** comienza con el comando `rintro` para introducir las variables `m` y `n`, y luego `⟨k, hk⟩` se utiliza para establecer la hipótesis de paridad `n = k + k`, donde `k` es un número natural y `hk` es una prueba de esta igualdad.

A continuación, se utiliza `use m * k` para establecer `m * k` como el número que demostrará que `m * n` es par.

Luego se utiliza `rw [hk]` para reemplazar `n` en la meta por `k + k`, utilizando la prueba `hk`.

Finalmente, se utiliza `exact mul_add m k k` para establecer que `m * (k + k)` es igual a `m * k + m * k`. Esto se logra utilizando la propiedad distributiva de la multiplicación respecto a la adición en los números naturales.

En resumen, la séptima demostración utiliza los comandos `rintro`, `use`, `rw` y `exact` para introducir variables, establecer el número testigo, realizar reemplazos y proporcionar una prueba final que demuestra la paridad de `m * n`.

La **8ª demostración** comienza con el comando `intros m n hn` para introducir las variables `m` y `n`, así como la hipótesis de paridad `hn`. Luego se utiliza `unfold Even at *` para desplegar la definición de paridad en todos los lugares relevantes.

A continuación, se utiliza `cases hn with | intro k hk =>` para realizar un análisis de casos sobre la hipótesis de paridad `hn`. En el caso en que `hn` se cumple y se puede demostrar que `n = k + k`, se introduce una nueva variable `k` y una prueba `hk` que establece esa igualdad.

Dentro de este caso, se utiliza `use m * k` para establecer `m * k` como el número que demostrará que `m * n` es par. Luego se utiliza `rw [hk, mul_add]` para realizar los reemplazos correspondientes.

En resumen, la octava demostración utiliza `intros` para introducir las variables `m`, `n` y la hipótesis de paridad `hn`. Luego se utiliza `unfold Even at *` para desplegar la definición de paridad en todos los lugares relevantes. A continuación, se realiza un análisis de casos sobre la hipótesis de paridad utilizando `cases`, y se introduce el número testigo utilizando `use`. Finalmente, se realiza el reemplazo utilizando `rw` para simplificar la expresión y demostrar la paridad de `m * n`.

La **9ª demostración** comienza con el comando `intros m n hn` para introducir las variables `m` y `n`, así como la hipótesis de paridad `hn`. Luego se utiliza `unfold Even at *` para desplegar la definición de paridad en todos los lugares relevantes.

A continuación, se utiliza `cases hn with | intro k hk =>` para realizar un análisis de casos sobre la hipótesis de paridad `hn`. En el caso en que hn se cumple y se puede demostrar que `n = k + k`, se introduce una nueva variable

`k` y una prueba `hk` que establece esa igualdad.

Dentro de este caso, se utiliza `use m * k` para establecer `m * k` como el número que demostrará que `m * n` es par.

Luego, se utiliza `calc m * n = m * (k + k) := by exact congrArg (HMul.hMul m) hk` para realizar un razonamiento algebraico paso a paso. La igualdad se deriva aplicando el lema `congrArg` al valor `m` y la prueba `hk` para mostrar que la multiplicación preserva la igualdad. Esto establece que `m * n` es igual a `m * (k + k)`.

Finalmente, se utiliza `_ = m * k + m * k := by exact mul_add m k k` para aplicar el lema `mul_add` y establecer que `m * k + m * k` es igual a `m * (k + k)`. Esto se logra utilizando la propiedad distributiva de la multiplicación respecto a la adición en los números naturales.

En resumen, la novena demostración utiliza `intros` para introducir las variables `m`, `n` y la hipótesis de paridad `hn`. Luego se utiliza `unfold Even at *` para desplegar la definición de paridad en todos los lugares relevantes. A continuación, se realiza un análisis de casos sobre la hipótesis de paridad utilizando `cases`, se introduce el número testigo utilizando `use`, y se utiliza `calc` y `by` para realizar razonamientos algebraicos paso a paso y establecer la igualdad necesaria.

La **10ª demostración** utiliza una estrategia de simplificación (`simp`) con las expresiones `*`, `parity_simps`. El comando `intros` se utiliza para introducir las variables y se utiliza `;` para combinar múltiples comandos en una sola línea.

El `simp` se aplica a las expresiones `*` y `parity_simps`. La expresión `*` indica que se deben aplicar simplificaciones con el contexto y la meta, mientras que `parity_simps` indica que se deben aplicar simplificaciones específicas relacionadas con la paridad de los números naturales.

En resumen, la décima demostración utiliza `intros` para introducir las variablesy luego aplica el comando `simp` con `*`, `parity_simps` para realizar las simplificaciones necesarias en la expresión `m * n` y demostrar su paridad. Esta estrategia simplificada permite una demostración concisa y automática del resultado deseado.

En **resumen**, todas las demostraciones presentadas son válidas y demuestran la misma proposición. Algunas utilizan tácticas más directas y simples, mientras que otras exploran diferentes enfoques y estilos de escritura en Lean.

Capítulo 2

Aspectos básicos del razonamiento matemático en Lean

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

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

2.1. Cálculos

2.1.1. Asociativa conmutativa de los reales

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

import Mathlib.Tactic
import Mathlib.Data.Real.Basic

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

```
example (a b c : ℝ) : (a * b) * c = b * (a * c) := by
  rw [mul_comm a b]
  rw [mul_assoc b a c]

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

-- El desarrollo de la prueba es:
--
--    inicio
--       a b c : ℝ
--       ⊢ (a * b) * c = b * (a * c)
--    rw [mul_comm a b]
--       a b c : ℝ
--       ⊢ (a * b) * c = b * (a * c)
--    rw [mul_assoc b a c]
--       goals accomplished

-- 2ª demostración
-- ===============

example
  (a b c : ℝ)
  : (a * b) * c = b * (a * c) :=
calc
  (a * b) * c = (b * a) * c := by rw [mul_comm a b]
            _ = b * (a * c) := by rw [mul_assoc b a c]

-- Comentario: El entorno calc permite escribir demostraciones
-- ecuacionales.

-- 3ª demostración
-- ===============

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

-- Comentario: La táctica ring demuestra ecuaciones aplicando las
```

```
-- propiedades de anillos.
```

Demostraciones comentadas

En estas demostraciones, se muestra que para cualquier número real `a`, `b` y `c`, se cumple la igualdad `(a * b) * c = b * (a * c)`. A continuación, se explica cada demostración en detalle:

En la **1ª demostración** se utiliza la táctica `by` junto con la táctica `rw` (`rewrite`) para reescribir la expresión y llegar a la igualdad deseada.

La táctica `rw [mul_comm a b]` se utiliza para aplicar la conmutatividad de la multiplicación y cambiar el orden de `a` y `b` en la expresión `(a * b)`. Después, la táctica `rw [mul_assoc b a c]` se utiliza para aplicar la asociatividad de la multiplicación y reagrupar los términos de la expresión `((b * a) * c)` en `(b * (a * c))`.

Al combinar estas dos tácticas, se reescribe la expresión original hasta llegar a la igualdad deseada.

En **2ª demostración** se utiliza la táctica `calc` para realizar una cadena de igualdades y llegar a la igualdad deseada.

La cadena de igualdades comienza con `(a * b) * c` y se utiliza la táctica `rw [mul_comm a b]` para reescribir `(a * b)` como `(b * a)`. Luego, se utiliza la táctica `rw [mul_assoc b a c]` para reescribir `(b * a) * c` como `b * (a * c)`.

Al utilizar la táctica calc de esta manera, se muestra paso a paso cómo se llega a la igualdad deseada a través de una cadena de reescrituras.

En la **3ª demostración** se utiliza la táctica `by ring` para demostrar la igualdad directamente utilizando propiedades algebraicas conocidas.

La táctica `by ring` se utiliza cuando se trabaja con anillos, como en este caso con los números reales. Esta táctica aplica automáticamente las reglas algebraicas básicas, como la conmutatividad y la asociatividad de la multiplicación, para simplificar y demostrar la igualdad.

En este caso, la táctica `by ring` reorganiza automáticamente los términos en la expresión `(a * b) * c` y llega a la forma `b * (a * c)`, demostrando así la igualdad.

En **resumen**, estas demostraciones muestran diferentes enfoques para demostrar la igualdad `(a * b) * c = b * (a * c)` utilizando tácticas de reescritura y propiedades algebraicas básicas. Cada demostración presenta un enfoque distinto, pero todos llegan al mismo resultado.

2.1.2. Ejercicio sobre aritmética real (1)

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

import Mathlib.Tactic
import Mathlib.Data.Real.Basic

-- 1ª demostración
-- ===============

example
  (a b c : ℝ)
  : (c * b) * a = b * (a * c) :=
by
  rw [mul_comm c b]
  rw [mul_assoc]
  rw [mul_comm c a]

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

--    a b c : ℝ
--    ⊢ (c * b) * a = b * (a * c)
-- rw [mul_comm c b]
--    a b c : ℝ
--    ⊢ (b * c) * a = b * (a * c)
-- rw [mul_assoc]
--    a b c : ℝ
--    ⊢ b * (c * a) = b * (a * c)
-- rw [mul_comm c a]
--    goals accomplished

-- 2ª demostración
-- ===============

example
  (a b c : ℝ)
  : (c * b) * a = b * (a * c) :=
calc
  (c * b) * a
    = (b * c) * a := by rw [mul_comm c b]
```

```
  _ = b * (c * a) := by rw [mul_assoc]
  _ = b * (a * c) := by rw [mul_comm c a]

-- 3ª demostración
-- ===============

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

Demostraciones comentadas

Las tres demostraciones son variantes equivalentes para demostrar la misma igualdad, utilizando tácticas diferentes. Voy a explicar cada una de ellas en detalle:

En la **1ª demostración** se utiliza la táctica `rw` para realizar reescrituras. La igualdad que se quiere demostrar es `(c * b) * a = b * (a * c)`. La demostración comienza con `by`, que indica que se utilizarán tácticas para completar la prueba.

Luego, se utiliza `rw [mul_comm c b]`, que aplica la regla de reescritura para intercambiar `c` y `b` en la expresión `(c * b) * a`, obteniendo así `(b * c) * a`.

A continuación, se utiliza `rw [mul_assoc]`, que aplica la regla de asociatividad de la multiplicación para reagrupar los términos, obteniendo `b * (c * a)`.

Finalmente, se utiliza `rw [mul_comm c a]`, que aplica la regla de reescritura para intercambiar `c` y `a` en la expresión `b * (c * a)`, obteniendo así `b * (a * c)`. La prueba se considera completada y se ha demostrado la igualdad deseada.

En la **2ª demostración** se utiliza la táctica `calc` para realizar cálculos sucesivos. La igualdad que se quiere demostrar es `(c * b) * a = b * (a * c)`.

La prueba comienza con `(c * b) * a`, y utilizando `:=` se establece que es igual a `(b * c) * a`. Esto se logra mediante `by rw [mul_comm c b]`, que aplica la regla de reescritura para intercambiar `c` y `b`.

A continuación, se utiliza `_ =` para indicar que el resultado actual es igual a `b * (c * a)`. Esto se logra mediante `by rw [mul_assoc]`, que aplica la regla de asociatividad de la multiplicación.

Finalmente, se utiliza `_ =` nuevamente para indicar que el resultado actual es igual a `b * (a * c)`. Esto se logra mediante `by rw [mul_comm c a]`, que

aplica la regla de reescritura para intercambiar `c` y `a`. La prueba se considera completada y se ha demostrado la igualdad deseada.

En la **3ª demostración** se utiliza la táctica `ring` para demostrar la igualdad automáticamente. La táctica `ring` es capaz de manejar expresiones algebraicas y aplicar reglas de simplificación y reescritura para demostrar igualdades.

La prueba comienza con `by ring`, que indica que se utilizará la táctica ring para completar la prueba. Esta táctica analiza la expresión `(c * b) * a` y la iguala automáticamente a `b * (a * c)` aplicando las reglas algebraicas necesarias.

La táctica `ring` es muy útil para demostrar igualdades algebraicas simples de forma automática, sin necesidad de especificar pasos intermedios. En este caso, la igualdad se demuestra de manera automática y la prueba se considera completada.

En **resumen**, las tres demostraciones son equivalentes y demuestran la igualdad `(c * b) * a = b * (a * c)` utilizando tácticas diferentes: reescrituras (`rw`), cálculos sucesivos (`calc`), y la táctica automática `ring`.

2.1.3. Ejercicio sobre aritmética real (2)

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

import Mathlib.Tactic
import Mathlib.Data.Real.Basic

-- 1ª demostración
-- ===============

example
  (a b c : ℝ)
  : a * (b * c) = b * (a * c) :=
by
  rw [←mul_assoc]
  rw [mul_comm a b]
  rw [mul_assoc]

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

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

--    a b c : ℝ
--    ⊢ a * (b * c) = b * (a * c)
-- rw [←mul_assoc]
--    a b c : ℝ
--    ⊢ (a * b) * c = b * (a * c)
-- rw [mul_comm a b]
--    a b c : ℝ
--    ⊢ (b * a) * c = b * (a * c)
-- rw [mul_assoc]
--    goals accomplished

-- 2ª demostración
-- ===============

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

-- 3ª demostración
-- ===============

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

Demostraciones comentadas

Las tres demostraciones son formas diferentes de demostrar la igualdad `a * (b * c) = b * (a * c)`, donde `a`, `b` y `c` son números reales.

En la **1ª demostración**, se utilizan las tácticas de reescritura (`rw`) para manipular la expresión y llegar al resultado deseado. La demostración se realiza en un bloque `by`, lo que significa que todas las tácticas se aplican secuencialmente. Aquí se muestra el paso a paso:

1. Se utiliza la táctica `rw [←mul_assoc]` para reescribir la expresión `a * (b * c)` como `(a * b) * c`. Esto se hace utilizando la asociatividad de la multiplicación.

2. Luego, se utiliza `rw [mul_comm a b]` para reescribir la expresión `a * b` como `b * a`. Esto se hace utilizando la conmutatividad de la multiplicación.

3. Finalmente, se utiliza `rw [mul_assoc]` para reescribir la expresión `(b * a) * c` como `b * (a * c)`. Nuevamente, se aplica la asociatividad de la multiplicación.

Al seguir estos pasos, se llega a la igualdad deseada: `a * (b * c) = b * (a * c)`.

En la **2ª demostración**, se utiliza la táctica `calc` para realizar la demostración utilizando un estilo más conciso y estructurado. Aquí se muestra el paso a paso:

1. Se inicia con la expresión `a * (b * c)`.

2. Luego, se utiliza la táctica `by rw [←mul_assoc]` para reescribir la expresión como `(a * b) * c`. Esto se hace utilizando la asociatividad de la multiplicación.

3. A continuación, se utiliza la táctica `by rw [mul_comm a b]` para reescribir la expresión como `(b * a) * c`. Esto se hace utilizando la conmutatividad de la multiplicación.

4. Por último, se utiliza la táctica `by rw [mul_assoc]` para reescribir la expresión como `b * (a * c)`. Se aplica la asociatividad de la multiplicación nuevamente.

Al seguir estos pasos, se llega a la igualdad deseada: `a * (b * c) = b * (a * c)`.

En la **3ª demostración**, se utiliza la táctica `ring` para demostrar la igualdad. La táctica `ring` es una táctica poderosa que puede demostrar automáticamente muchas identidades algebraicas.

Al utilizar `by ring`, se le indica al sistema de demostración automática que pruebe la igualdad utilizando propiedades algebraicas. En este caso, el sistema reconoce automáticamente que se puede aplicar la conmutatividad y la asociatividad de la multiplicación para llegar al resultado deseado.

En **resumen**, las tres demostraciones utilizan diferentes tácticas y estilos para llegar a la igualdad `a * (b * c) = b * (a * c)`. La primera y segunda demostración utilizan las tácticas `rw` y `calc`, respectivamente, para reescribir la expresión paso a paso. La tercera demostración utiliza la táctica `ring` para demostrar automáticamente la igualdad utilizando propiedades algebraicas.

2.1.4. Ejemplo de rw con hipótesis

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

import Mathlib.Tactic
import Mathlib.Data.Real.Basic

-- 1ª demostración
-- ===============

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

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

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

-- inicio
--    a b c d e f : ℝ,
--    h1 : a * b = c * d,
--    h2 : e = f
--    ⊢ a * (b * e) = c * (d * f)
-- rw [h2]
--    S
--    ⊢ a * (b * f) = c * (d * f)
-- rw [←mul_assoc]
--    S
--    ⊢ (a * b) * f = c * (d * f)
```

```
-- rw [h1]
--    S
--    ⊢ (c * d) * f = c * (d * f)
-- rw [mul_assoc]
--    goals accomplished
--
-- En el desarrollo anterior, S es el conjunto de las hipótesis; es
-- decir,
--    S = {a b c d e f : ℝ,
--         h1 : a * b = c * d,
--         h2 : e = f}

-- 2ª demostración
-- ===============

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

-- 3ª demostración
-- ===============

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

2.1.4.1. Demostraciones comentadas

Las tres demostraciones presentadas tienen como objetivo demostrar la igualdad: `a * (b * e) = c * (d * f)`, utilizando las hipótesis `h1: a * b = c * d` y `h2: e = f`. A continuación, comentaré cada una de las demostraciones:

En la **1ª demostración** se utiliza el enfoque de reescribir (`rw`) expresiones utilizando las igualdades dadas. El primer paso es reemplazar `e` por `f` usando la hipótesis `h2` (`rw [h2]`). Luego, se utiliza el lema de asociatividad de la multiplicación en sentido inverso (`←mul_assoc`) para reorganizar los términos y obtener `(a * b) * f = (c * d) * f`. Por último, se utiliza la hipótesis `h1` (`rw [h1]`) para reemplazar `a * b` por `c * d` y, finalmente, usando la asociatividad se llega a la igualdad deseada.

En la **2ª demostración** se utiliza el enfoque de cálculo (`calc`) para realizar una secuencia de pasos de igualdad. Comienza con~a * (b * e)~ y se utiliza la hipótesis `h2` para reemplazar `e` por `f` (`by rw [h2]`). Luego, se utiliza el lema de asociatividad de la multiplicación en sentido inverso (`←mul_assoc`) para reorganizar los términos y obtener `a * (b * f) = (a * b) * f`. A continuación, se utiliza la hipótesis `h1` para reemplazar `a * b` por `c * d` (`by rw [h1]`) y se obtiene `(c * d) * f`. Finalmente, se utiliza nuevamente el lema de asociatividad de la multiplicación en sentido directo (`mul_assoc`) para reorganizar los términos y obtener `c * (d * f)`, llegando así a la igualdad deseada.

En la **3ª demostración** se utiliza el enfoque de simplificación (`simp`). Se utiliza el modificador * para indicar que se deben utilizar todas las hipótesis y lemas disponibles. En este caso, se utiliza * y `←mul_assoc` para aplicar el lema de asociatividad de la multiplicación en sentido inverso. El objetivo es simplificar `a * (b * e)` a `c * (d * f)` directamente, aprovechando las igualdades `h1` y `h2`. Este enfoque permite simplificar la demostración a una sola línea.

En **resumen**, las tres demostraciones logran el mismo objetivo de demostrar la igualdad `a * (b * e) = c * (d * f)` utilizando diferentes enfoques. La primera utiliza reescrituras explícitas (`rw`), la segunda utiliza el enfoque de cálculo (`calc`) y la tercera utiliza la simplificación automática (`simp`). Cada enfoque tiene sus propias ventajas y puede ser preferido dependiendo del contexto y de la experiencia del desarrollador.

Capítulo 3

Bibliografía

- Lean 4 cheatsheet. ~ Martin Dvořák.
- Lean 4 manual.
- Mathematics in Lean. ~ Jeremy Avigad y Patrick Massot.
- Reference sheet for people who know Lean 3 and want to write tactic-based proofs in Lean 4. ~ Martin Dvořák.
- Theorem proving in Lean 4. ~ Jeremy Avigad, Leonardo de Moura, Soonho Kong y Sebastian Ullrich.