TFG

TRABAJO FIN DE GRADO
Curso 2024/2025

UNIVERSIDAD COMPLUTENSE
MADRID

FACULTAD DE CIENCIAS MATEMÁTICAS

GRADO EN MATEMÁTICAS

DEPARTAMENTO DE SISTEMAS INFORMÁTICOS Y COMPUTACIÓN

Nombre del estudiante: Longjian Jiang
Nombre del tutor: Jorge Carmona Ruber

Madrid, 9 de julio de 2025

Abstract

This thesis is about formalizing mathematical proofs using LEAN4. The proof of the Inverse Function Theorem in $\mathbb{R}$ will be the main example to illustrate the formalization.

Índice general

INTRODUCCIÓN

En matemáticas, para verificar una demostración puede ser muy costosa. Con lo que aparició el problema de demostrar los teoremas de forma automática utilizando ordenadores. LEAN4 es un asistente de pruebas interactivo y un lenguaje de programación funcional diseñado para dicho propósito, es decir, una vez que pruebemos el resultado en LEAN4, no tenemos que preocupar por su veracidad, ya que está garantizada. Además, con el rápido avance de la Inteligencia Artificial, surgió la ambición de que la demostración pueda ser automática.

El presente trabajo trata de introducir el marco teórico en el que se basa LEAN4, y unos ejemplos prácticos de cómo formalizar resultados matemáticos con la ayuda de LEAN4. El principal resultado será el Teorema de la función inversa en $\mathbb{R}$. Para ello, es conveniente familizarse con el lenguaje LEAN4 y con la librería Mathlib. Asimismo, en muchas ocasiones, hay dos formas de hacer pruebas, una de ellas consiste usar las tácticas (teoremas probados) existentes en Mathlib, y otra posibilidad sería demostrar el resultado elaborarando nuestras propias definiciones. Cabe destacar una tercera posibilidad mezclando las dos formas anteriores.

En cuanto a la estructura, el documento comenzará con un capítulo dedicado a explicar los fundamentos del lenguaje LEAN4

Capítulo 1

Los objetivos y el plan de trabajo

El presente trabajo trata de comprender cómo usar LEAN4 para probar resultados matemáticos. Para ello, vamos a ver en el siguiente capítulo la base teórica de LEAN4. Tras haber adquirido cierta base teórica, se presenta unos ejercicios simples realizados con LEAN4 para ilustrar la teoría expuesta. Más adelante, se muestra numerosas tacticas importantes para las demostraciones realizadas.

Posteriormente, se encuentra un capítulo en el que trata de analizar un ejemplo realizado por el autor durante el curso, que es más complejo. En concreto, consiste en demostrar el Teorema de la Función Inversa para $\mathbb{R}$. Además, la demostración está hecha de dos maneras, una utilizando las tácticas, y otra bansandose en nuestras propias definiciones que requiere probar la equivlancia con las de Mathlib.

Asimismo, se va a explicar las diferencias entre los dos métodos, y las conexiones que hay entre ellos.

Por último, se reflexiona sobre todo el contenido de la presente memoria y lo trabajado durante el curso. De esta manera, se concluye el proyecto.

Capítulo 2

El lenguaje LEAN4

Para poder enlazar con las tareas prácticas, este capítulo se centra en explicar la base teórica y presentar unas tácticas fundamentales. Además, se ilustrará con algunos ejemplos simples para facilitar la comprensión.

2.1. Fundamento teórico

LEAN4 se basa en una versión de teoría de tipos dependientes llamado calculo de construcciones, con una jerarquía contable de universos no cumulativos y tipos inductivos. Para comprender mejor su significado, veamos las siguientes nociones.

2.1.1. Teoría de tipos

La teoría de tipos clasifica los objetos según su tipo. Por ejemplo, en el siguiente contexto, **X** representa un número real y **F** una función de $\mathbb{R}$ en $\mathbb{R}$.

```
def X : Real := 2.3
def F : Real → Real := λ x ⇒ 2 * x
```

Ejemplo 2.1: Declaración de variables en Lean 4.

Cabe destacar que en LEAN, los propios tipos son objetos, y cada uno de ellos tiene un tipo.

```
def a : Type := Nat
def b : Type := Bool
#check a    -- Type
#check b    -- Type
```

Ejemplo 2.2: Declaración de tipos como objetos en Lean4.

El comando #check pregunta a LEAN4 de qué tipo es el objeto, y el comando #eval hace la evaluación de la expresión dada. Además, podemos construir nuevos tipos a partir de otros, por ejemplo, tenemos dos tipos a y b, podemos construir el nuevo tipo de función a → b. De esta manera, si consideramos ***Type 0*** como el universo de los tipos "pequeños", ***Type 1*** como un universo más grande que contiene a ***Type 0*** como un elemento, y ***Type 2*** el universo más grande que contenga a ***Type 1*** como un elemento. Reiterando el proceso, obtenemos una lista infinita de universos de tipos en la que para cada $n \in \mathbb{N}$, existe un ***Type n***, y se sigue la jerarquía de que ***Type n+1*** es un universo más grande que contiene a ***Type n*** como un elemento.

En cuanto a la abstracción de funciones y su evaluación, LEAN4 se apoya en λ-cálculo. Si tenemos una variable $\boldsymbol{x}$ **:** α, y construimos una expresión $\boldsymbol{t}$ **:** β, tendríamos la función fun (x : α) $\Rightarrow$ t, o de forma equivalente, $\lambda\ (x : \alpha) \Rightarrow t$, que también es un objeto de tipo $\alpha \to \beta$.

```
def f ( n : Nat ) : Nat := 2*n
def g ( m : Nat ) : Bool := m % 2 = 0

#check fun x : Nat ⇒ g (f x)    -- Nat → Bool
#check fun x ⇒ g (f x)          -- Nat → Bool
```

Ejemplo 2.3: Abstracción de funciones en Lean4.

Observemos que para definir la composición de f y g, no es necesario precisar el tipo de x, puesto que LEAN4 lo infiere automáticamente a partir de las definiciones de f y g. Si intentamos hacer la composición de f (g x), se producirá un error, y LEAN4 nos proporciona información sobre el fallo como indica la siguiente figura.

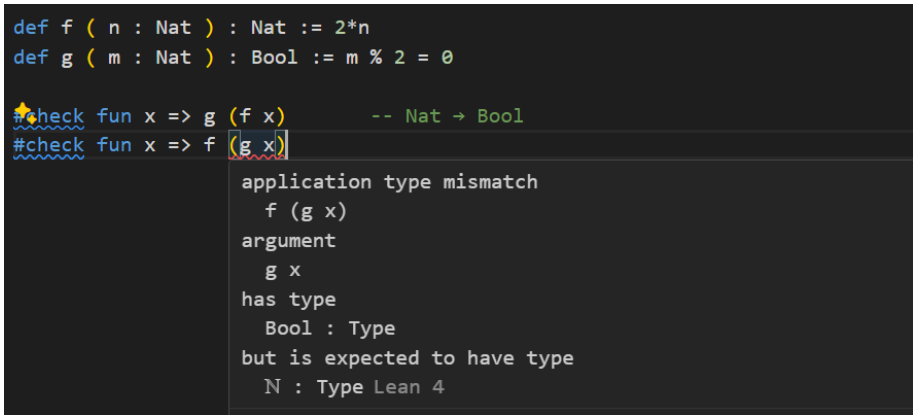

Figura 2.1: Error por tipos inadecuados en Lean4.

Los tipos pueden depender de los argumentos, por ejemplo ***List*** α depende del parámetro α, por ello ***List Nat*** se distingue de ***List Bool***.

```
def concat (A : Type) (as bs : List A ) : List A :=
  as ++ bs

#check concat Nat  -- List Nat → List Nat → List Nat
#check concat Real -- List Real → List Real → List Real
#check concat      -- (A : Type) → List A → List A → List A
```

Ejemplo 2.4: Dependencia de tipos.

El ejemplo anterior define la función que concatena dos listas de mismo tipo, y se trata de una función dependiente de tipos. En el que podemos ver la dependencia de la función por el parámetro de entrada.

2.1.2. Tipos inductivos

Un tipo indcutivo se construye con una lista de constructores, y cada constructor puede tomar argumentos que pueden ser del mismo tipo inductivo, permitiendo así la recursión. Se trata de una herramienta muy potente, en la que todos los tipos que no sean el universo de tipos o tipo de función dependiente, se puede construir como un tipo inductivo.

```
inductive Season where
  | spring : Season
  | summer : Season
  | fall : Season
  | winter : Season
```

Ejemplo 2.5: Ejemplo simple de tipo inductivo finito.

Además, al definir un tipo inductivo, podemos usar ***deriving Repr*** para que LEAN4 pueda generar una función que convierte los objetos ***Season*** en texto. Se puede visualizarlo en tipos-inductivos.

Para ilustrar la recursión, vamos a definir los naturales con dos construtctores.

```
inductive Nat where
    | zero : Nat
    | succ : Nat → Nat
```

Ejemplo 2.6: Definición del tipo Nat.

El primer constructor ***zero : Nat*** no toma ningún argumento, y el constructor ***succ*** requiere la previa construcción de ***Nat***. Aplicando ***succ*** a ***zero*** obtenemos un nuevo elemento de tipo ***Nat***, y de forma reiterada conseguimos reproducir todos los números naturales.

2.2. Proposiciones

En LEAN4, se introduce el tipo ***Prop*** para representar las proposiciones, y se introducen unos constructores para poder construir proposiciones a partir de otras (la conjunción, la disyunción, la negación y la implicación). Además, para cada proposición ***p : Prop***, se introduce el tipo ***Proof p***. Para el tipo ***Proof p***, un axioma sería un constante.

```
def Implies (p q : Prop) : Prop := p → q
#check And     -- Prop → Prop → Prop
#check Or      -- Prop → Prop → Prop
#check Not     -- Prop → Prop
#check Implies -- Prop → Prop → Prop

variable (p q r : Prop)
#check And p q                      -- Prop
#check Or (And p q) r               -- Prop
#check Implies (And p q) (And q p)  -- Prop
```

Ejemplo 2.7: Constructores de proposiciones.

2.3. Los teoremas en LEAN4

En esta sección, vamos a ver como está estructurado un teorema.

```
theorem name
    decl1 ... decln :
    result := by
    proof
```

Figura 2.2: Estructrua de un teorema en LEAN4.

- name es el nombre que le damos al teorema.
- decl1 ... decln son las declaraciones que tenemos para obtener el resultado, que pueden ser de distintos tipos, y lo ilustramos a continuación con un ejemplo.
- result denota el resultado que se obtiene a partir de las declaraciones que tenemos.Además, siempre es de tipo Prop.
- proof es la prueba del teorema, que normalmente se basa en subdividir el objetivo en piezas más pequeñas y probar estas.

```
theorem min_lt_min                                                    source
      {α : Type u} [LinearOrder α] {a b c d : α} (h₁ : a < c)
      (h₂ : b < d) :
    min a b < min c d
```

Ejemplo 2.8: Ejemplo ilustrativo de la estructura de los teoremas en LEAN4.

En el ejemplo anterior, tiene 6 declaraciones, la primera es una variable α que nos indica el tipo que va a tener los elementos que se van a usar posteriormente. La segunda declaración dice que el tipo α pertenece a la clase LinearOrder. La tercera declaración corresponde a que los cuatro elementos a,b,c y d que vamos a usar son del tipo α. Las dos últimas declaraciones son dos hipótesis que tenemos sobre los cuatro elementos anteriores.

2.4. Las principales tácticas

En esta sección, vamos a presentar unas tácticas fundamentales para demostrar resultados en LEAN4.

La táctica intro se usa para manejar implicaciones, negaciones y cuantificadores universales. Su función principal es descomponer el objetivo en hipótesis o variables en el contexto local.

```
example (p q : Prop) (h1 : p = False): p → q := by
  intro h     --El objetivo aún queda pendiente
```

Ejemplo 2.8: Ejemplo del uso de intro.

Tras haber usado intro h, se añade una nueva hipótesis h : p al conjunto de hipótesis que tenemos, dejando el objetivo como q.

La táctica exact sirve para probar el objetivo cuando lo tenemos ya como una hipótesis.

```
example (x : Real) (h : x > 0) : x > 0 := by
  exact h     --Objetivo probado
```

Ejemplo 2.9: Ejemplo del uso de exact.

La táctica rw su nombre se trata de la abreviatura de rewrite", por lo que su función consiste en reescribir cuando tenemos una expresión de tipo a = b o a ↔ b. Hay varias maneras de uso, reescribir a por b o b por a, sobre el objetivo o sobre alguna hipotesis. Al poner rw [expresion] , se intentra reescribir a por b sobre el objetivo, y rw [← expresion] para reescribir b por a. Si después de rw [...] ponemos at h, se reescribe sobre la hipótesis h.

```
example (x y: Real) (h1 : x > 0) (h2 : y = 0) : x > y := by
  rw [y]  --Tras este comando, el objetivo se queda como x>0
  exact h1

example (x y z : Real) (h1 : x = z) (h2 : y = z) :
  x = y := by
  rw [← h2] at h1   --Deja a h1 como x = y
  exact h1
```

Ejemplo 2.9: Ejemplo del uso de rw.

Las táctica unfold reemplaza una expresión por su definición.

```
def zero : Nat := 0
example (n : Nat):
    n + zero = n := by
    unfold zero
    apply add_zero
```

Ejemplo 2.12: Ejemplo del uso de unfold.

La táctica apply se usa para aplicar una hipótesis o teorema sobre el objetvio o alguna hipótesis que tenemos.

```
example (n1 n2 n3 n4 : Nat) (h1 : n1 < n3) (h2: n2 < n4) :
    min n1 n2 < min n3 n4 := by
    apply min_lt_min  --Prueba el objetivo original, y crea
                      --dos nuevos objetivos que eran las
                      --hipotesis del teorema

example (n1 n2 n3 n4 : Nat) (h1 : n1 < n3) (h2: n2 < n4) :
    min n1 n2 < min n3 n4 := by
    apply min_lt_min h1 h2  --Deja todo probado, pues h1 h2
                            --son las hipotesis lo que
                            --min_lt_min exige

example ( p q r : Prop) (h1 : p → r ) (h2 : r → q) :
    p → q := by
    intro h          --Extraemos h: p como hipótesis
    apply h1 at h    --Deja h como h : r
    apply h2 at h    --Deja h como h : q
    exact h
```

Ejemplo 2.10: Ejemplo del uso de apply.

La táctica constructor sirve para separar los objetivos del tipo $p \wedge q$ en dos objetivos p y q. Para los objetivos de tipo $p \leftrightarrow q$ también se puede usar constructor, ya que no deja de ser $p \to q \wedge q \to p$.

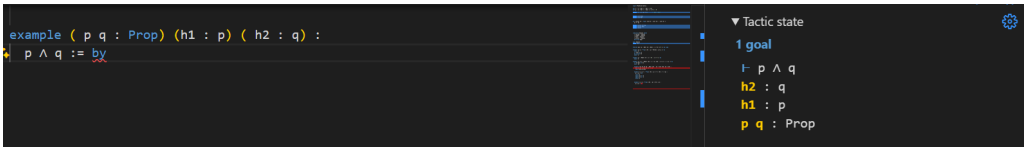

Figura 2.3: Ejemplo antes de ejecutar constructor.

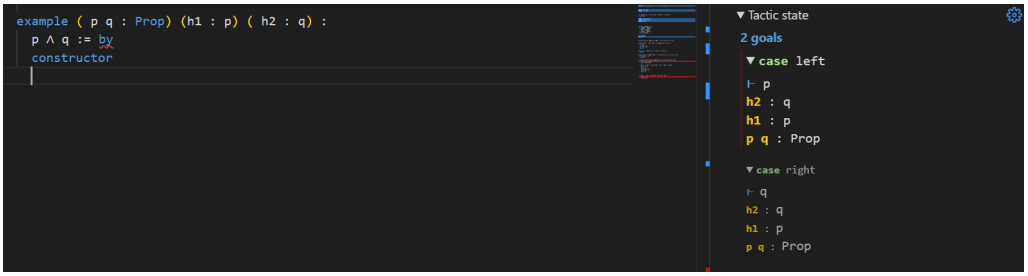

Figura 2.4: Ejemplo después de ejecutar constructor.

Las dos figuras anteriores ilustran cómo transforma el objetivo en dos subojetivos más simples tras el uso de constructor.

Las tácticas left y right se usan para extraer "la parte izquierdaz "la parte derecha"del objetivo cuando se trata de una disyunción, o sirven para extraer su correspondiente parte de una hipótesis cuando esta es de forma $p \wedge q$.

```
example ( p q : Prop) (h1 : p ∧ q) ( h2 : q) :
  p ∨ q := by
  left
  exact h1.left
```

Ejemplo 2.11: Ejemplo de uso de left y right.

Tras aplicr left, el objetivo sólo será p, y h1.left extrae p de la hipótesis h1: $p \wedge q$.

Las tácticas use y obtain se usan principalmente para cuantifadores existenciales.La táctica use es de uso exclusivo para los objetivos, en el que dado un objetivo de tipo $\exists x$, p x, proporcionamos un x concreto que cumpla el predicado p. En cambio, obtain sirve para extraer un elemento x que satisfaga el predicado p, si tenemos a $\exists x$, p x como hipótesis.

```
example (f : Real → Real) (hf: ∃ x : Real, f x = 0 ) :
  ∃ x : Real, f x + 1 = 1 := by
  obtain ⟨ x, hx ⟩ := hf
  use x
  rw [hx]
  apply zero_add
```

Ejemplo 2.13: Ejemplo del uso de use y obtain

Las táctica specialize se utiliza para aplicar a una hipótesis con un cuantificador universal.

```
example (f : Real → Real) (hf: ∀ x : Real, f x = 0 ):
  f 2 = 0 := by
  specialize hf 2
  exact hf
```

Ejemplo 2.14: Ejemplo del uso de specialize

La táctica have sirve para introducir nuevas hipótesis, las cuales han de ser probadas. Con esta táctica, se puede dividir el problema en subproblemas más sencillos, e ir probando uno por uno.

```
example (f : Real → Real) (hf: ∀ x : Real, f x = 0 ) ( y z: Real) :
  y * f z = 0 := by
  have f_zero : f z = 0 := by
    specialize hf z
    exact hf
  rw [f_zero]
  rw [mul_zero]
```

Ejemplo 2.14: Ejemplo del uso de have

Las tácticas cases' y by_cases se utilizan para hacer distinción de casos, la diferencia entre ellos consiste en que cases' se usa para una hipótesis ya existente, y by_cases permite crear una nueva hipótesis para probar el objetivo actual, pero se necesitará probar otra vez el objetivo con la negación de la nueva hipótesis que habíamos creado.

```
example (x : ℝ) (h: (x = 1) ∨ (x = 2)) :
  x > 0 := by
  cases' h with h1 h2 --Tenemos que probar dos veces el objetivo
  rw [h1]             --Probar con hipotesis h1
  linarith

  rw [h2]             --Probar con h2
  linarith

example (f: ℝ → ℝ )(x : ℝ) (h1 : (x = 0) → f x = 1) (h2: (x ≠ 0) → f x > 3 ):
  f x > 0 := by
  by_cases x_eq_0 : x = 0    --Suponemos que x = 0
  apply h1 at x_eq_0
  linarith

  apply h2 at x_eq_0         --Caso en el que x ≠ 0
  linarith
```

Ejemplo 2.15: Ejemplo del uso de cases' y by_cases

Además de las tácticas anteriores, hay otras muy interesantes que se van a exponer de palabra. Comenzamos con ***by_contra***, que nos permite transformar la negación del objetivo como hipótesis, y el objetivo que queda por probar será ***False***, es decir, esta táctica nos permite probar resultados por contradicción.

Otro comando interesante sería ***let***, que nos permite definir expresiones u objetos de forma local, en otras palabras, ***let*** define un término dentro de un bloque que ayuda a simplificar la prueba, por ejemplo, si se tiene una expresión en la que se repite varias veces $3 + 2$, podemos hacer let k := $2 + 3$, y probar el objetivo con k.

En el ejemplo 2.15, se ha utilizado la táctica linarith cuya función es resolver objetivos relacionados con desigualdades lineales y ecuaciones aritméticas básicas. En el caso del ejemplo, prueba que $3 > 0$ y $1 > 0$. Otra herramienta de gran utilidad sería ***simp***, que nos permite simplificar expresiones en el caso de que exista un regla predefinida para ello.

Por último, sorry es un comando que nos permite omitir la prueba sin que el sistema produzca errores, pero LEAN4 también nos avisa de que la prueba aún está sin terminar.