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 apareció 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, se puede construir nuevos tipos a partir de otros, por ejemplo, sean dos tipos a y b, se puede construir el nuevo tipo de función a $\rightarrow$ b. De esta manera, si se considera ***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 se tiene una variable $\boldsymbol{x : \alpha}$, y se construye una expresión $\boldsymbol{t : \beta}$ a partir de la cual, se puede obtener la función $\boldsymbol{fun\ (\ x : \alpha\ ) \Rightarrow t}$, 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.

Obrsérvese 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 se intenta hacer la composición de f (g x), se producirá un error, y LEAN4 proporciona información sobre el fallo como indica la siguiente figura.

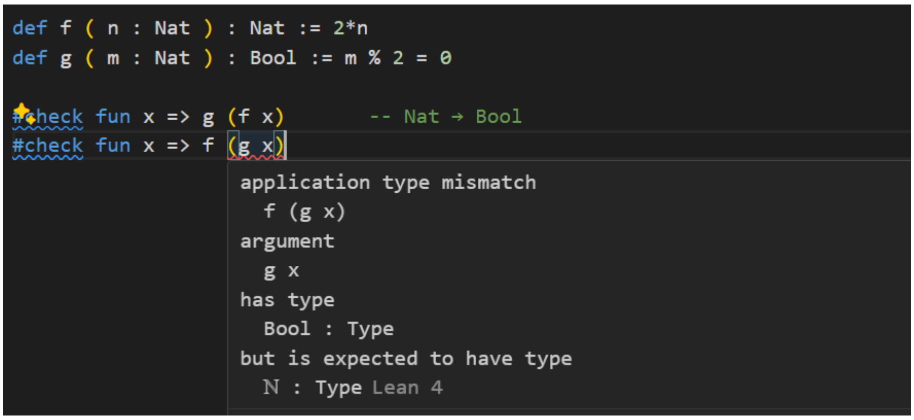

Figura 2.1: Error por tipos inadecuados en Lean4.

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 cons (α : Type) (a : α) (as : List α) : List α :=
  List.cons a as

#check cons Nat          -- Nat → List Nat → List Nat
#check cons Bool         -- Bool → List Bool → List Bool
#check cons              -- (α : Type) → α → List α → List α
```

Ejemplo 2.4: Dependencia de tipos.

El ejemplo anterior define la función que inserta un nuevo elemento al principio de una lista dada, y se trata de una función dependiente de tipos. En el que dados $\alpha : Type$ y $\beta : \alpha \to Type$, donde para cada $a : \alpha$, $\beta\ a$ es un tipo. Además, el tipo $(a : \alpha) \to \beta\ a$ denota el tipo de funciones $\boldsymbol{f}$ en el que para cada $a : \alpha$, $f\ a$ es un elemento de $\beta\ a$. Es decir, el tipo del valor que devuelve f depende de los parámetros de entrada, en el que refleja la dependencia de la función por el input.

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, se puede 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, se definen los naturales con dos construtctores.

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

def add (m n : Nat) : Nat :=
    match n with
    | Nat.zero ⇒ m
    | Nat.succ n ⇒ Nat.succ (add m n)

open Nat

#eval add (succ (succ zero)) (succ zero) --Nat.succ (Nat.succ (Nat.succ
                                         --(Nat.zero)))
```

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*** se obtiene un nuevo elemento de tipo ***Nat***, y de forma reiterada se consigue reproducir todos los números naturales.

Además, se puede definir la suma de los números naturales m y n con recursión sobre n. En el caso base, si n es zero, se devuelve el valor m. Suponiendo que se conoce el valor de ***add m n***, se define ***add m (succ n)*** como ***succ (add m n)***.

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, se va a presentar 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 se tiene para obtener el resultado, que pueden ser de distintos tipos, y se ilustr 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 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 se va a usar son del tipo α. Las dos últimas declaraciones son dos hipótesis que se tiene sobre los cuatro elementos anteriores.

2.4. Las principales tácticas

En esta sección, se presentan 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 viene de la abreviatura de *rewrite*, por lo que su función consiste en reescribir cuando se tiene una expresión de tipo $a = b$ o $a \leftrightarrow b$. 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 [...] se pone ***at h***, se reescribe sobre la hipótesis h.

```
example (x y: Real) (h1 : x > 0) (h2 : y = 0) : x > y := by
  rw [h2]  --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 se tiene.

```
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 $p \rightarrow q \wedge q \rightarrow p$.

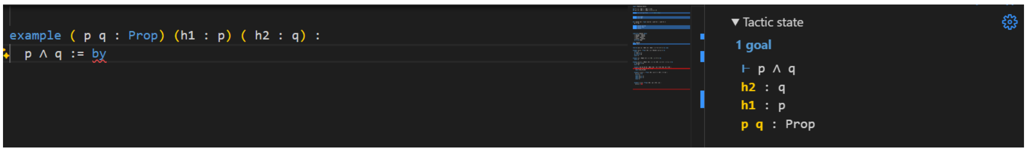

Figura 2.3: Ejemplo antes de ejecutar constructor.

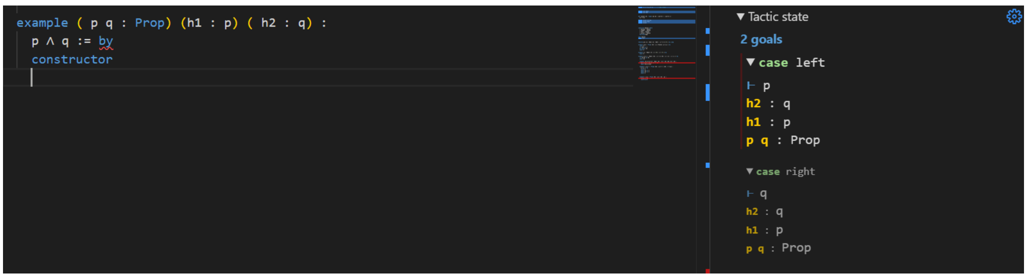

Figura 2.4: Ejemplo después de ejecutar constructor.

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 $\exists x, p\ x$, se proporciona 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. Cabe destacar que, cuando se tiene una hipótesis de tipo $\exists x, p\ x$, puede existir varios elementos que cumplan p, por lo que la elección puede no ser única. Cuando sólo es de interés abstraer matemáticamente su existencia, no es un problema. Pero cuando se requiere una construcción explícita, ***obtain*** deja de ser tan útil, ya que requerirá ver también la equivalencia entre el cumplimiento para un elemento y el cumplimiento para el otro que puede ser distinto.

```
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. Se comienza con ***by_contra***, que permite transformar la negación del objetivo como hipótesis, y el objetivo que queda por probar será ***False***, es decir, esta táctica permite probar resultados por contradicción.

Otro comando interesante sería ***let***, que 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, se puede hacer ***let k := 3 + 2***, 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 permite simplificar expresiones en el caso de que exista un regla predefinida para ello.

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

Capítulo 3

Teorema de la función inversa en $\mathbb{R}$ demostrado en LEAN4

En este capítulo, se presenta la demostración del Teorema de la función inversa en $\mathbb{R}$. Además, está realizada por el autor de dos maneras, la primera consiste en utilizar las tácticas y los teoremas ya existentes en la librería Mathlib. La otra forma se trata de elaborar unas definiciones, probar la equivalencia de estas con las definiciones de LEAN4, y probar el resultado utilizando dichas definiciones. Esta última tiene especial interés debido a que facillita la comprensión de la demostración, aunque su proceos es mucho más costoso que el modo táctico. Además, en matemáticas es frecuente que ciertas nociones estén definidas de distintas maneras, y para probar ciertos resultados relacionados con dichas nociones, conviene usar una definición que otras, pero para que se pueda comprender con más produndidad la prueba, se dedica un esfuerzo para ver la equivalencia entre las definiciones.

En primer lugar, se presenta cómo está estructurado el problema en LEAN4.

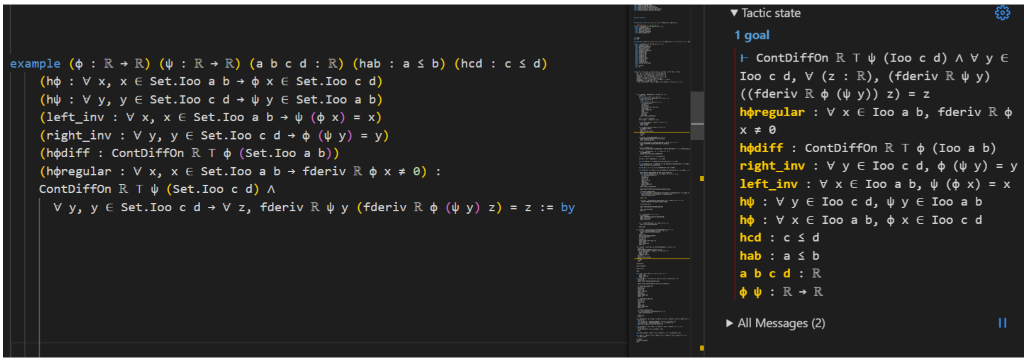

Figura 3.1: Formulación del TFI en $\mathbb{R}$ en LEAN4.

3.1. Modo táctico

En esta sección, se trata de explicar cómo se ha conseguido probar el resultado utilizando las tácticas y los teoremas ya existentes de Mathlib (librería de LEAN4). Para ello, se introducen las definiciones y teoremas necesarios para comprender y resolver el problema. Las cuales están hechas con la máxima generalidad posible. Posteriormente, en la sección de probar el resultado con propias definiciones elaboradas se podrá ver con más claridad, estableciendo una conexión con las nociones del lenguaje matemático usual para este caso particular.

3.1.1. Definiciones

```
def ContDiffOn                                                          source
        (𝕜 : Type u) [NontriviallyNormedField 𝕜] {E : Type uE}
        [NormedAddCommGroup E] [NormedSpace 𝕜 E] {F : Type uF}
        [NormedAddCommGroup F] [NormedSpace 𝕜 F] (n : WithTop ℕ∞) (f : E → F)
        (s : Set E) :
    Prop
```

A function is continuously differentiable up to `n` on `s` if, for any point `x` in `s`, it admits continuous derivatives up to order `n` on a neighborhood of `x` in `s`.

For `n = ∞`, we only require that this holds up to any finite order (where the neighborhood may depend on the finite order we consider).

▼ Equations

- `ContDiffOn 𝕜 n f s = ∀ x ∈ s, ContDiffWithinAt 𝕜 n f s x`

Def. 3.1: Definición de ContDiffOn en LEAN4.

En primer lugar, se tiene la definición de ***ContDiffOn*** en el que toma 3 parámetros explícitos, y otros implícitos que son exigencias sobre los espacios que se van a usar. Tras recibir los 3 parámetros, determina si la función ***f*** es continuamente diferenciable de orden ***n*** en el conjunto ***s***. Por otro lado, está la definición de ContDiffAt en la que no toma el parámetro del conjunto, sino un punto, y determina si la función es continuamente diferenciable de orden n en el punto dado.

```
@[irreducible]                                                          source
def fderiv
        (𝕜 : Type u_4) [NontriviallyNormedField 𝕜] {E : Type u_5} [AddCommGroup E]
        [Module 𝕜 E] [TopologicalSpace E] {F : Type u_6} [AddCommGroup F]
        [Module 𝕜 F] [TopologicalSpace F] (f : E → F) (x : E) :
    E →L[𝕜] F
```

If `f` has a derivative at `x`, then `fderiv 𝕜 f x` is such a derivative. Otherwise, it is set to `0`.

▼ Equations

- `fderiv 𝕜 f x = fderivWithin 𝕜 f Set.univ x`

Def. 3.2: Definición de fderiv en LEAN4.

Además de ***fderiv***, que se trata de la función derivada de una función en un punto dado, en Mathlib está la definición de ***deriv*** que consiste en el valor de la función derivada en dicho punto. Asimismo, en Mathlib está predefinida una regla que permite relacionar ***fderiv*** con ***deriv*** utilizando el comando ***simp***.

```
@[simp]                                                                 source
theorem fderiv_eq_smul_deriv
        {𝕜 : Type u} [NontriviallyNormedField 𝕜] {F : Type v}
        [NormedAddCommGroup F] [NormedSpace 𝕜 F] {f : 𝕜 → F} {x : 𝕜} (y : 𝕜) :
    (fderiv 𝕜 f x) y = y • deriv f x
```

Figura. 3.2: Relación entre deriv y fderiv en LEAN4.

Las definiciones anteriores son imprescindibles para entender el problema, más adelante se precisarán otras definiciones en el momento que se considere oportuno para la comprensión de la demostración.

3.1.2. La demostración del problema

A continuación, se va a exponer los teoremas ya existentes en Mathlib que sirven para probar el resultado, y explicar cómo hacer uso de dichos teoremas. Se puede ver la demostración completa del problema con tácticas en TFI_IN_R_TACTICS.

```
theorem ContDiffAt.to_localInverse                                          source
        {𝕂 : Type u_1} [RCLike 𝕂] {E : Type u_2} [NormedAddCommGroup E]
        [NormedSpace 𝕂 E] {F : Type u_3} [NormedAddCommGroup F] [NormedSpace 𝕂 F]
        [CompleteSpace E] {f : E → F} {f' : E ≃L[𝕂] F} {a : E} {n : WithTop ℕ∞}
        (hf : ContDiffAt 𝕂 n f a) (hf' : HasFDerivAt f (↑f') a) (hn : 1 ≤ n) :
    ContDiffAt 𝕂 n (hf.localInverse hf' hn) (f a)
```

Given a `ContDiff` function over `𝕂` (which is `ℝ` or `ℂ`) with an invertible derivative at `a`, the inverse function (produced by `ContDiff.toPartialHomeomorph`) is also `ContDiff`.

Teorema 3.1: Teorema principal que se va a usar.

Como el problema se trata probar resultados en intervalos abiertos, la noción de ser ***ContdiffOn*** en un conjunto equivale a ser ***ContDiffAt*** en todos los puntos del conjunto. Además, para probar dicha equivalencia, en Mathlib hay un teorema contdiffOn_iff que trata de resolver dicha equivalencia.

Por lo anterior, se puede reducir la parte de diferenciabilidad continua en un conjunto abierto como un caso particular del *Teorema 3.1*. Para poder apicar este teorema, se ha de reunir las hipótesis que exige el teorema, y probar que efectivamente la función ψ coincide la inversa local de ϕ en el intervalo abierto. De esta manera, la estrategia que se va a seguir es utilizar la táctica have para introducir estas hipótesis, es decir, con la ayuda del teorema basta probar subproblemas más sencillos.

```
example (φ : ℝ → ℝ) (ψ : ℝ → ℝ) ... :
    ... := by      --La parte de puntos suspensivos se puede ver en la
                   --figura 3.1
    have ψ_smooth : ContDiffOn ℝ ⊤ ψ (Ioo c d) := by
      have open_inter : IsOpen (Ioo c d) := by exact isOpen_Ioo
      have open_inter' : IsOpen (Ioo a b) := by exact isOpen_Ioo
      rw [IsOpen.contDiffOn_iff open_inter]
      rw [IsOpen.contDiffOn_iff open_inter'] at hφdiff
      intro y hy
      have hx : ψ y ∈ Ioo a b := hψ y hy
      specialize hφdiff hx -- A hφdiff lo especifica con ψ y
       ...            --La parte omitida puede verse en el enlace
                      --de la prueba completa
      have hf' : HasFDerivAt φ (φ' : ℝ →L[ℝ] ℝ) (ψ y) := by
         ...
      have hn : 1 ≤ (⊤ : WithTop ℕ∞)  := by simp
```

Figura 3.3: Prueba de diferenciabilidad cotinua en subproblemas más sencillos

En la figura anterior, ya se ha conseguido reunir las hipótesis para aplicar el Teorema 3.1, la hipótesis ***hϕdiff*** tras una serie de aplicaciones de comandos y teoremas, se queda como ***hϕdiff : ContDiffAt*** $\mathbb{R}$ $\top$ ϕ *(ψ y)* . Por tanto, tras aplicar el teorema con las correspondientes hipótesis, se tiene que la inversa local es infinitamente diferenciable en ϕ (ψ y). Como $y \in$ *Ioo c d* , por la hipótesis ***right_inverse***, se tiene que ϕ (ψ y) es igual a y. De este modo, sólo queda pendiente probar que la función ψ coincide con la inversa local en el intervalo abierto. Obsérvese que no es posible probar la igual entre esas dos funciones, ya que sólo se puede garantizar la igualdad en el intervalo abierto a partir de las hipótesis ***right_inverse*** y ***left_inverse***. Por ello, requiere hacer uso del siguiente teorema para concluir la diferenciabilidad de ψ, además de ver que coincide con la inversa local en el intervalo abierto.

```
theorem ContDiffAt.congr_of_eventuallyEq                                    source
        {𝕜 : Type u} [NontriviallyNormedField 𝕜] {E : Type uE}
        [NormedAddCommGroup E] [NormedSpace 𝕜 E] {F : Type uF}
        [NormedAddCommGroup F] [NormedSpace 𝕜 F] {f f₁ : E → F} {x : E}
        {n : WithTop ℕ∞} (h : ContDiffAt 𝕜 n f x) (hg : f₁ =ᶠ[nhds x] f) :
    ContDiffAt 𝕜 n f₁ x
```

Teorema 3.2: Teorema auxiliar para la diferenciabilidad.

```
def Filter.EventuallyEq                                                     source
        {α : Type u_1} {β : Type u_2} (l : Filter α) (f g : α → β) :
    Prop
```

Two functions `f` and `g` are *eventually equal* along a filter `l` if the set of `x` such that `f x = g x` belongs to `l`.

▼ Equations

- `(f =ᶠ[l] g) = ∀ᶠ (x : α) in l, f x = g x`

▶ Instances For

Def.3.3: Definición de la igualdad eventual.

La Def. 3.3 corresponde a la noción de que dos funciones coinciden localmente, y es una relación que se puede probar entre ψ y la función inversa local de ϕ en los puntos del intervalo abierto $Ioo\ a\ b$. Por tanto, se puede usar el Teorema 3.2 para probar la diferenciabilidad continua de ψ al tener las hipótesis de que la inversa local es continuamente diferenciable, y las dos funciones coinciden localmente.

```
    have local_inv: ψ =ᶠ[nhds (φ (ψ y))] (hφdiff.localInverse hf' hn) := by
        ...     --La parte omitida se puede ver con el enlace

    apply right_inv at hy
    rw [← hy]
    have inv_theorem := ContDiffAt.to_localInverse hφdiff hf' hn
    apply ContDiffAt.congr_of_eventuallyEq inv_theorem local_inv
```

Figura 3.4: La continuación de la figura 3.3

Tras combinar estos resultados, queda probado la parte de diferenciabilidad. Observese que no se ha aplicado el comando constructor al principio para probar como objetivo, sino se ha introducido como una hipótesis. La ventaja de hacer esto es que dicha hipótesis puede ser usada posteriormente, si se procede de dicha manera, tras probarlo como objetivo, no se puede usar o que no se tiene como hipótesis para la prueba de la segunda parte.

```
constructor
exact ψ_smooth
```

Figura 3.5: Cierre de la prueba de diferenciabilidad.

La segunda parte de la demostración consiste principalmente en utilizar la regla de la cadena, con lo que se va a usar el siguiente teorema de Mathlib.

```
theorem deriv_comp                                                          source
        {𝕜 : Type u} [NontriviallyNormedField 𝕜] (x : 𝕜) {𝕜' : Type u_1}
        [NontriviallyNormedField 𝕜'] [NormedAlgebra 𝕜 𝕜'] {h : 𝕜 → 𝕜'}
        {h₂ : 𝕜' → 𝕜'} (hh₂ : DifferentiableAt 𝕜' h₂ (h x))
        (hh : DifferentiableAt 𝕜 h x) :
    deriv (h₂ ∘ h) x = deriv h₂ (h x) * deriv h x
```

Teorema.3.3: Teorema principal para probar la parte de derivada.

Utilizando la relación entre fderiv y deriv (véase Figura 3.2), se puede transformar el problema en términos de deriv.

```
intro y hy z
simp
have h_id : ∀ y ∈ Ioo c d, φ (ψ y) = id y := by
    ...    --La parte omitida puede verse en la demostración completa
have h_id_deriv : ∀ y ∈ Ioo c d, deriv (φ ∘ ψ) y = deriv id y := by
    ...
```

Figura 3.6: Inicio de la prueba de la segunda parte.

Con el código adjunto, se consigue que el problema esté en términos de deriv. Además, combinando el Teorema 3.3 con el Teorema deriv_id, se puede demostrar el resultado por completo. Pero antes de usar dichos resultados, es necesarior probar la diferenciabilidad de las dos funciones, es por ello se ha tomado la parte de ser continuamente diferenciable infinitamente como hipótesis, ya que se puede extraer la diferenciabilidad de ser infinitamente difereniable de manera sencilla.

```
have hψ_diff : ∀ y ∈ Ioo c d, DifferentiableAt ℝ ψ y := by
    have aux: DifferentiableOn ℝ ψ (Ioo c d) := by
        apply ψ_smooth.differentiableOn
        simp
    intro y hy
    apply aux.differentiableAt
    rw [mem_nhds_iff]
    use Ioo c d
    constructor
    exact subset_refl (Ioo c d)
    constructor
    exact isOpen_Ioo
    exact hy

have hφ_diff : ∀ y ∈ Ioo c d, DifferentiableAt ℝ φ (ψ y) := by
    intro y hy
    apply differentiableAt_of_deriv_ne_zero
    have hx_fder_nonzero : fderiv ℝ φ (ψ y) ≠ 0 := by
        apply hφregular
        apply hψ y hy
    rw [←fderiv_deriv]
    intro h1
    apply hx_fder_nonzero
    ext t
    rw [h1]
    simp
```

Figura 3.7: Desarrollo de la prueba de la segunda parte.

En la figura anterior se muestra la prueba de las condiciones de diferenciabilidad, aunque se trata de ejercicios simples, es interesante ver que se puede probar un mismo resultado de distintas maneras. La primera hipótesis se ha probado basándose en la diferenciabilidad infinita, es fácil de ver que se puede probar de manera análoga para la segunda hipótesis. Pero se ha optado por probar que tiene la función derivada no nula, puesto que en LEAN4 las funciones que no son derivables, se considera por defecto que su derivada es 0.

```
have chain_deriv : ∀ y ∈ Ioo c d,
                   deriv (φ ∘ ψ) y = (deriv φ (ψ y)) * (deriv ψ y) := by
      intro y hy
      have hφ_diff_at : DifferentiableAt ℝ φ (ψ y) := hφ_diff y hy
      have hψ_diff_at : DifferentiableAt ℝ ψ y := hψ_diff y hy
      apply deriv_comp y hφ_diff_at hψ_diff_at

have h_id_deriv1 : ∀ y ∈ Ioo c d, (deriv φ (ψ y)) * (deriv ψ y) = 1 := by
      intro y hy
      rw [← chain_deriv y hy, h_id_deriv y hy]
      simp

have deriv_product : deriv φ (ψ y) * deriv ψ y = 1 := h_id_deriv1 y hy

have asoc: z * deriv φ (ψ y) * deriv ψ y = z * (deriv φ (ψ y) * deriv ψ y)
:= by
      rw [mul_assoc]

rw [asoc]
rw [deriv_product]
simp
```

Figura 3.9: Cierre de la prueba de la segunda parte.

Tras haber obtenido la diferenciabilidad de las dos funciones, se puede aplicar el Teorema 3.3 para obtener la igualdad entre la multiplicación de dos derivadas y la derivada de la compuesta. Además, combinando la hipótesis ***h_id_deriv*** y el Teorema ***deiv_id*** se consigue verificar que la multiplicación de las dos derivadas vale uno en todo el intervalo ***Ioo c d***. Tras tener dichos resultados, es suficiente utilizar la asociatividad y la conmutatividad para cerrar esta sgunda parte.

3.2. Elaborarando definiciones

En esta sección se presentan las definiciones elaboradas por el autor, las cuales están basadas en términos de ϵ y δ. Es decir, las nociones se definen utilizando el lenguaje matemático más usual. Además, se demuestra la equivalencia entre estas definiciones con las de Mathlib.

3.2.1. Personalizar definiciones para la segunda parte

En primer lugar, se definen los conceptos relacionados con la segunda parte del problema. Cabe destacar que en LEAN4, las definiciones son totales, un ejemplo sería el caso de ***deriv***, en el cual aunque la función no es derivable en cierto punto, se considera por defecto que su ***deriv*** es 0 en el punto dado. Por ello, la definición personalizada se adapatará a ello, para que se cumpla la equivlaencia.

```
def Myhasder  (f : ℝ → ℝ) (f' x  : ℝ) : Prop :=
  ∀ ε > 0, ∃ δ > 0, ∀ y, 0 < |y - x| ∧ |y - x| < δ
  → |(f (y) - f x) / (y-x) - f'| < ε
```

Figura 3.10: Definición elaborada por el autor.

Para poder definir ***deriv***, es necesario definir unas nociones previamente. ***Myhasder*** corresponde a la definición ***HasDerivAt*** de Mathlib. Toma 3 parámetros, una función ***f***, y dos números reales ***f'*** y ***x*** . En cuanto al significado, se refiere a que el valor de la derivada de ***f*** en el punto ***x*** es ***f'***.

```
def Mydiff (f : ℝ → ℝ) (x : ℝ) : Prop :=
  ∃ (f' : ℝ ), Myhasder f f' x

def Mycont (f : ℝ → ℝ) (x : ℝ) : Prop :=
  ∀ ε > 0, ∃ δ > 0, ∀ y, |y - x| < δ   →  |f y - f x| < ε
```

Figura 3.11: Definiciones elaboradas por el autor.

La definición ***Mydiff*** es equivalente a ***DifferentiableAt***, y ***Mycont*** corresponde a la definición de ***ContinuousAt***. Teniendo estas definiciones, se puede definir ***deriv***.

```
open Classical in
noncomputable def Myderiv (f : ℝ → ℝ) (x: ℝ) : ℝ  :=
  if h: Mydiff f x  then
    Classical.choose h
  else
    0
```

Figura 3.12: Definición elaborada por el autor.

El comando ***open Classical in*** permite acceder a los resultados de la lógica clásica, que por defecto no está disponible en LEAN4. Además, al tratarse de una definición que depende de axiomas no computables, se ha de añadir ***noncomputable*** antes de la definición para que no salten errores. El motivo de que falla la compilación en el caso de no poner ***noncomputable***, se debe a que la definición contiene elementos no ejecutables (no se puede obtener un resultado concreto al ejecutarlos, pero sirve para demostraciones de otros resultados).

En cuanto al contenido de la definición, evalua si la función es diferenciable en el punto dado, al estar ***Mydiff*** definido con el cuantificador existencial, es decir, de forma $h : \exists\ (x : \alpha),\ p\ x$, lo que hace ***Classical.choose*** es devolver un elemento que satisface p, y a diferencia de ***obtain***, independientemente de si existen varios elementos que satisfagan el predicado, la elección es única. Es remarcable que se tiene la unicidad debido a que ***Classical.choose*** está basado en el axioma de elección. En el caso de que no se diferenciable, se devuelve el valor 0 para lograr la totalidad de la definición.

Una vez definidas las nociones necesarias, se han de probar que estas coinciden con las Mathlib, es decir, hay que demostrar la equivalencia entre las personalizadas y las predefinidas. Cabe destacar que este ejercicio es análogo a lo que se hace en las matemáticas, en el que se caracteriza un concepto en múltiples definiciones, siendo algunas más útiles que otras bajo determinados contextos.

A continuación, se prensentan algunas de las demostraciones de las equivalencias hechas por el autor.

```
theorem cont (f : ℝ → ℝ) (x : ℝ):
  ContinuousAt f x ↔ Mycont f x := by
  constructor    --La implicación →
  intro h
  unfold Mycont
  rw [Metric.continuousAt_iff] at h
  intro ε hε
  obtain ⟨ δ, δ_pos, hδ ⟩ := h ε hε
  use δ
  constructor
  exact δ_pos
  intro y hy
```

```
  apply hδ at hy
  rw [Real.dist_eq] at hy
  exact hy

  intro h       --La implicación ←
  rw [Metric.continuousAt_iff]
  unfold Mycont at h
  intro ε hε
  obtain ⟨ δ, δ_pos, hδ ⟩ := h ε hε
  use δ
  constructor
  exact δ_pos
  intro y hy
  apply hδ at hy
  rw [Real.dist_eq]
  exact hy
```

Figura 3.13: Equivalencia entre ContnuousAt y Mycont.

Se comienza con el ejemplo más simple. El primer paso es aplicar el comando ***constructor*** para dividir la equivalencia en dos implicaciones, la primera de ellas corresponde al caso $\Rightarrow$, y la segunda $\Leftarrow$. Ambas implicaciones se basan en el teorema ***Metric.continuousAt_iff*** de Mathlib, en el que expresa la continuidad en un punto en función de las distancias. Como en el caso de $\mathbb{R}$, la noción de distancia coincide con el valor absoluto entre la diferencia de dos números, las implicaciones se consiguen de manera sencilla haciendo uso de ***use*** y ***obtain***.

```
theorem has_der (f : ℝ → ℝ) (f' x  : ℝ) :
  HasDerivAt f f' x ↔ Myhasder f f' x := by
  constructor
  intro h        --La implicación →
  unfold Myhasder
  rw [hasDerivAt_iff_tendsto] at h
  intro ε hε
  rcases (Metric.tendsto_nhds_nhds.1 h) ε hε with ⟨δ, hδ_pos, hδ⟩
  use δ

  constructor
  exact hδ_pos  --Prueba que δ > 0

  intro y hy
  cases' hy with hy1 hy2
  apply hδ at hy2
  rw [Real.dist_eq] at hy2
  simp at hy2
  rw [← abs_inv] at hy2
  rw [← abs_mul] at hy2
  rw [abs_abs,inv_mul_eq_div, sub_div, mul_comm,
  ← mul_div,div_self, mul_one] at hy2
  exact hy2     --prueba que  y - x ≠ 0
  rw [← abs_ne_zero]
  rw [← abs_pos]
  rw [abs_abs]
  exact hy1

  intro h     --La implicación ←
  rw [hasDerivAt_iff_tendsto]
  rw [Metric.tendsto_nhds_nhds]
  intro ε hε
  obtain ⟨ δ , δ_pos, hδ  ⟩ := h ε hε
  use δ
```

```
  constructor
  exact δ_pos    --Prueba que δ > 0
  intro z hz
  rw [Real.dist_eq]
  simp
  rw [← abs_inv,← abs_mul,abs_abs,inv_mul_eq_div,
  sub_div, mul_comm,← mul_div]
  by_cases hxz_eq : z = x    --Se distinguen dos casos
  rw [hxz_eq]                --Caso z = x
  simp
  exact hε                   --Prueba trivial de ϵ > 0
  have hxz_des : z - x ≠ 0 := by
    rw [sub_ne_zero]
    exact hxz_eq

  rw [← abs_ne_zero] at hxz_des
  rw [← abs_pos] at hxz_des
  rw [abs_abs] at hxz_des
  specialize hδ z
  have aux: 0 < |z - x| ∧ |z - x| < δ  := by
    constructor
    exact hxz_des
    rw [Real.dist_eq] at hz
    exact hz

  apply hδ at aux          --aux queda en forma del objetivo
  rw [div_self, mul_one]   --Al aplicar div_self, hace falta probar
                           --también que el divisor es no nulo.
  exact aux
  rw [← abs_ne_zero]
  linarith                 --Prueba de divisor no nulo
```

Figura 3.14: Equivalencia entre las HasDerivAt y Myhasder.

Esta demostración es más complicada que la anterior en el sentido de que hace falta distinguir casos para poder probar el objetivo. Especialmente en la parte de $\Leftarrow$, se ha de distinguir el caso de si z coincide con x, puesto que en la definición personalizada no se tiene en cuenta este caso para evitar división sobre cero (lo que se hace en el lenguaje matemático usual). Sin embargo, al tener la totalidad de las funciones en LEAN4, y la división por cero toma por defecto el valor cero, el objetivo se reduce trivialmente a $\epsilon > 0$.

Además, cabe destacar los teoremas que se han aplicado para poder transformar una desigualdad en otra que es equivalente. Aunque se trata de una tarea pesada, es interesante ver la cantidad de resultados que se aplican en las operaciones que se consideran obvias.

```
theorem diff (f : ℝ → ℝ) (x : ℝ) :
  DifferentiableAt ℝ f x ↔ Mydiff f x := by
  constructor
  intro h          --La implicación →
  rw [DifferentiableAt] at h
  unfold Mydiff
  obtain ⟨ f', hf' ⟩ := h
  use (f' 1)             --Relación entre fderiv y deriv
  rw [← has_der]
  rw [hasFDerivAt_iff_hasDerivAt] at hf'
  exact hf'

  intro h           --La implicación ←
  rw [Mydiff] at h
  rw [DifferentiableAt]
```

```
  obtain ⟨ f', hf⟩  := h
  rw [← has_der] at hf
  rw [hasDerivAt_iff_hasFDerivAt] at hf
  let g : ℝ →L[ℝ] ℝ := ContinuousLinearMap.smulRight
  (ContinuousLinearMap.id ℝ ℝ) f'   --Relación entre fderiv y deriv
  use g
  exact hf
```

Figura 3.15: Equivalencia entre las DifferentiableAt y Mydiff.

Para demostrar esta equivalencia, se procede de manera similar a las anteriores. No obstante, al estar definida ***DifferentiableAt*** con la existencia de la función derivada, se ha de jugar la relación entre ***fderiv*** y ***deriv***.

Por último, se presenta la equivalencia entre ***deriv*** y ***Myderiv***.

```
theorem der_equiv (f : ℝ → ℝ) (x  : ℝ) :
  deriv f x = Myderiv f x := by
  by_cases h : ∃ f', HasDerivAt f f' x  --Caso en el que es derivable
  have h' : ∃ f', HasDerivAt f f' x := h
  rw [HasDerivAt.deriv]
  obtain ⟨ f' , hf'⟩ := h'
  have aux : Myderiv f x = f' := by
    unfold Myderiv
    split_ifs with h1
    unfold Mydiff at h1
    apply der_unique f _ f' x   --derv_unique se presenta posteriormente
    exact Classical.choose_spec h1
    rw [← has_der]
    exact hf'
    unfold Mydiff at h1
    by_contra h'
    rw [has_der] at hf'
    apply h1
    use f'

  rw [aux]
  exact hf'
  --Caso no derivable
  have equiv: (∀ (x_1 : ℝ), ¬Myhasder f x_1 x) ↔
  (∀ (x_1 : ℝ), ¬HasDerivAt f x_1 x ):= by
    constructor
    intro h z
    specialize h z
    rw [has_der]
    exact h
    intro h z
    specialize h z
    rw [← has_der]
    exact h
  have zero1: Myderiv f x = 0 := by
    unfold Myderiv
    rw [dif_neg]
    unfold Mydiff
    simp at *
    rw [equiv]
    exact h

  have zero2 : deriv f x = 0 := by
    have ne_diff :  ¬ DifferentiableAt ℝ f x := by
```

```
      rw [diff]
      unfold Mydiff
      simp
      simp at h
      rw [equiv]
      exact h
    apply deriv_zero_of_not_differentiableAt ne_diff

  rw [zero1,zero2]
```

Figura 3.16: Equivalencia entre las deriv y Myderiv.

En el caso de que la función sea diferenciable se trata de jugar con Classical.choose y que la derivada si existe, es única. Para el caso no diferenciable, se trata de ver que tendrán valor 0. En otras palabras, la prueba del segundo caso consiste principalmente en el manejo de if-else en LEAN4, ya que en este caso es aplicar la definición de ***deriv*** y ***Myderiv***, y tener en cuenta que no es diferenciable la función en el punto dado.

3.2.2. Prueba de la segunda parte

Para probar la segunda parte, primero se reudce el problema en términos de ***deriv***, para poder aplicar las definiciones personalizadas.

```
intro y hy z
simp
rw [mul_assoc]
have eq_one : deriv φ (ψ y) * deriv ψ y  = 1 := by
    have res : deriv φ (ψ y) = 1 / deriv ψ y := by
         ...       --Presentada posteriormente
    rw [res]
    let k : ℝ := deriv ψ y
    have aux:  k = deriv ψ y := by
        unfold k
        rfl
    have h_ne : k ≠ 0 := by
        have hy' : ψ y ∈ Ioo a b := by
            apply hψ at hy
            exact hy
        apply hφregular at hy'
        rw [← norm_ne_zero_iff] at hy'
        rw [← norm_deriv_eq_norm_fderiv] at hy'
        rw [Real.norm_eq_abs] at hy'
        rw [abs_ne_zero] at hy'
        rw [res] at hy'
        intro h
        simp at hy'
        apply hy'
        exact h

    rw [mul_comm,← aux,mul_one_div_cancel]
    exact h_ne
rw [eq_one]
simp
```

Figura 3.17: La demostración de la segunda parte.

En la figura anterior se presenta la demostración de la segunda parte del problema. Se trata de ver que el producto de las dos derivadas tiene valor 1, y la tarea principal es ver que una es la inversa multiplicativa de la otra. Tras conseguir el objetivo, sólo hace falta ver que *deriv* ψ y no se anula, y es una hipótesis fácil de extraer a partir de las hipótesis que se tenían.

A continuación, se presenta la parte fundamental de esta prueba, algunas partes están omitidas, y puede verlo con todo el detalle en (enlace por poner).

```
have res : deriv φ (ψ y) = 1 / deriv ψ y := by
    have aux : HasDerivAt ψ (deriv φ (ψ y))⁻¹ y := by
        ...      --Se presenta al final de esta sección.
    nth_rewrite 2 [der_equiv]
    unfold Myderiv
    rw [has_der] at aux
    rw [dif_pos]
    have unique_der : @Classical.choose ℝ (fun x ↦ Myhasder ψ x y) ?hc
    = (deriv φ (ψ y))⁻¹ := by
        apply der_unique ψ
        have h_ex : ∃ f', Myhasder ψ f' y := by
            use (deriv φ (ψ y))⁻¹
        exact Classical.choose_spec h_ex
        exact aux
    rw[unique_der]
    simp
    rw [Mydiff]
    use (deriv φ (ψ y))⁻¹
```

Figura 3.18: Desarrollo de la segunda parte.

Para poder probar ***res***, se ha de probar que ψ tiene a $(deriv\ \phi\ (\psi\ y))^{-1}$ como derivada en el punto y. Tras probar ello, es necesario emplear la definición personalizada de ***deriv***, y al estar definida con ***Classical.choose***, aunque su elección es única, hay que ver que lo que ha elegido es $(deriv\ \phi\ (\psi\ y))^{-1}$. Por ello, se ha tenido que definir el siguiente teorema.

```
theorem der_unique (f : ℝ → ℝ) ( f1' f2' x : ℝ)
(h0: Myhasder f f1' x) (h1: Myhasder f f2' x):
  f1' = f2'  := by
  rw [← has_der] at *
  apply HasDerivAt.unique h0 h1
```

Figura 3.19: Teorema auxiliar de la segunda parte.

Por último, se muestra cómo se ha probado ***aux*** utilizando las definiciones basadas en ϵ y δ.

```
have aux : HasDerivAt ψ (deriv φ (ψ y))⁻¹ y := by
    rw [has_der]
    have der_ne_zero : deriv φ (ψ y) ≠ 0 := by
        ...        --Parte omitida en el enlace
    have hf : HasDerivAt φ (deriv φ (ψ y)) (ψ y) := by
        ...        --Parte omitida en el enlace
    rw [has_der] at hf
    unfold Myhasder at *
    let dφ := deriv φ (ψ y)
    intro ε ε_pos
    have dφ_ne_zero : dφ ≠ 0 := der_ne_zero
    let ε' := min (|dφ|/2) (ε * dφ^2 / 2)
    have ε'_pos : 0 < ε' := by
        ...
    obtain ⟨δ', δ'_pos, hδ'⟩ := hf ε' ε'_pos
    have ψ_cont : ContinuousAt ψ y := by
        ...
    obtain ⟨η, η_pos, hη⟩ : ∃ η > 0, ∀ y₁, |y₁ - y| < η
    → |ψ y₁ - ψ y| < δ' := by
     ...
```

```
    let r := min (y - c) (d - y)
    have r_pos : r > 0 := by
      ...
    let δ := min r η
    have δ_pos : δ > 0 := by
      ...
    use δ
    constructor
    exact δ_pos
    intro y₁ hy₁
    rcases hy₁ with ⟨y₁_ne, y₁_close⟩
    have y₁_close' : |y₁ - y| < η := by
      ...
    have y₁_close'''' : |y₁ - y| < r := by
      ...
    have hψ_close : |ψ y₁ - ψ y| < δ' := hη y₁ y₁_close'
    have in_Ioo : y₁ ∈ Ioo c d := by
    by_cases hψ_eq : ψ y₁ = ψ y  --Caso en el que ψ y₁ = ψ y
    have : y₁ = y := by
        rw [← right_inv y hy, ← right_inv y₁]
        rw [hψ_eq]
        exact in_Ioo
    rw [this] at y₁_ne
    simp at y₁_ne
    --Caso en el que ψ y₁ ≠ ψ y
    have hdiff : |(φ (ψ y₁) - φ (ψ y)) / (ψ y₁ - ψ y) - dφ| < ε' := by
      ...
    have y₁_eq : y₁ = φ (ψ y₁) := by
      ...
    have y_eq : y = φ (ψ y) := by
      ...
    rw [← y_eq,  ← y₁_eq] at hdiff
    let r := (y₁ - y) / (ψ y₁ - ψ y) - dφ
    have r_bound : |r| < ε' := hdiff
    have dφ_r_ne_zero : dφ + r ≠ 0 := by
      ...
    have : (ψ y₁ - ψ y) / (y₁ - y) = 1 / (dφ + r) := by
      ...
    rw [this]
    have : |1 / (dφ + r) - dφ⁻¹| = |r| / |dφ * (dφ + r)| := by
    have dφ_eq : dφ = deriv φ (ψ y) := by simp
    rw [← dφ_eq]
    rw [this]
    have d1: |dφ + r| ≥ |dφ| - |r| := by
      ...
    have d2: |dφ + r| ≥ |dφ| - ε' := by linarith
    have d3: |dφ + r| ≥ |dφ|/2 := by
      ...
    have d4: |dφ * (dφ + r)| ≥ |dφ| * (|dφ|/2) := by
      ...
    have d5: |r| / |dφ * (dφ + r)| ≤ |r| / (|dφ| * (|dφ|/2)) := by
      ...
    have d6: |r| / (|dφ| * (|dφ|/2)) = 2 * |r| / |dφ|^2 := by ring
    rw [d6] at *
    have ε'_le : ε' ≤ ε * dφ^2 / 2 := min_le_right _ _
    have d7 : |r| / (|dφ| * (|dφ| / 2)) < ε' / ( dφ ^ 2 / 2) := by
      ...
    have hpos : dφ ^ 2 / 2 > 0 := by
      ...
    have key : |r| / |dφ * (dφ + r)| < ε' / (dφ^2 / 2) := by
```

```
        apply lt_of_lt_of_le' d7 d5
    have ε'_bound : ε' / (dφ^2 / 2) ≤ (ε * dφ^2 / 2) / (dφ^2 / 2) := by
        apply (div_le_div_iff_of_pos_right hpos).mpr ε'_le
    have ε'_simplified : (ε * dφ^2 / 2) / (dφ^2 / 2) = ε := by
        field_simp [hpos.ne']

    rw [ε'_simplified] at ε'_bound
    apply lt_of_le_of_lt' ε'_bound key
```

Figura 3.20: Desarrollo fundamental de la segunda parte.

En primer lugar, hay que destacar que en LEAN4, se puede utilizar el mismo nombre para distintas variables, mientras que las anteriores no se usarán más veces. Por ejemplo, durante la prueba, se ha definido dos veces r, y la primera ya no es útil tras haber obtenido unas hipótesis, pues a la hora de definir otro r, lo que hace LEAN4 es dar un nombre nuevo al r antiguo, añadiéndole una cruz. Otro aspecto que tener en cuenta es, al definir una hipótesis sin dar nombre, LEAN4 por defecto se le nombra como ***this***.

En cuanto al contenido de la demostración, se denota a *deriv* ϕ $(\psi\ y)$ como $d\phi$ para poder simplificar las expresiones posteriormente. Dado un ϵ arbitrario mayor que cero, se define un nuevo ϵ' para extraer la δ' de la hipótesis hf, es decir, de que ϕ tenga su derivada. Además, al ser ψ continua, se puede sacar un η tal que $\forall\ x \in \mathbb{R}, |x - y| < \eta\ \ \rightarrow |\psi\ x\ \ -\ \ \psi\ y| < \delta'$. Se define un r para garantizar luego que y_1 esté dentro del intervalo (c, d). Definiendo δ como el mínimo entre η y r, y se usa dicho δ para ver que se cumple la desigualdad de que ψ tiene la derivada que se espera tener.

Haciendo uso de la δ definida anteriormente, se puede conseguir las hipótesis como y_1_close', y_1_close'''' y $h\psi\_close$ que son de utilidad para obtener *hdiff*, y_1_eq e y_eq. Las últimas dos hipótesis permiten reescribir a *hdiff* en forma de $|(y_1\ \ -\ \ y)/(\psi\ y_1\ \ - \psi\ y)\ \ - d\phi| < \epsilon'$, y una vez que se tiene esta hipótesis, lo que queda por probar es una serie de cálculos.

Los cálculos se centran en probar que $|\frac{\psi\ y_1\ -\ \psi\ y}{y_1\ - y}\ \ - \frac{1}{d\phi}| = |\frac{|r|}{|d\phi * (d\phi\ +\ r)|}$ y acotarlo en una serie de desigualdades $\frac{|r|}{|d\phi * (d\phi\ +\ r)|} \leq \frac{2|r|}{d\phi^2} < \frac{2\epsilon'}{d\phi^2} \leq \epsilon$, donde la última desigualdad se extrae de la hipótesis ϵ'_le y la segunda deisgualdad viene dada por r_bound. Se puede ver con todo el detalle de cómo se han hecho las pruebas de estos cálculos en el enlace adjuntado.