Formalización de las matemáticas con Lean. Un caso de estudio: Resultados de Topología General.

TRABAJO DE FIN DE GRADO

Curso 2024/2025

UNIVERSIDAD
COMPLUTENSE
MADRID

Pepa Montero Jimena

Julio de 2025

Dirigido por Jorge Carmona Ruber

FACULTAD DE CIENCIAS MATEMÁTICAS
GRADO EN MATEMÁTICAS
DEPARTAMENTO DE SISTEMAS INFORMÁTICOS Y COMPUTACIÓN

Resumen

Este trabajo consiste en una exploración del sistema Lean 4 como asistente de demostración, mediante un caso de estudio: la formalización de resultados de topología general. El objetivo principal es profundizar en el aprendizaje de este sistema y obtener una visión crítica de las ventajas y los inconvenientes que presenta al escribir matemáticas. Se presenta un acercamiento general a la base fundacional del sistema, su sintaxis y su aplicación concreta a la topología. Finalmente, se detalla el proceso de formalización de un resultado complejo como es el Lema de Urysohn, junto con las conclusiones obtenidas del trabajo realizado.

Abstract

This project consists of an exploration of the Lean 4 system as a proof assistant through a case study: the formalisation of results in general topology. The main objective is to deepen the understanding of this system and to provide a critical perspective on its advantages and disadvantages when writing mathematics. A general overview of the foundational basis of the system, its syntax, and its concrete application to topology is provided. The project concludes with a description of the process of formalising the more complex result Urysohn's Lemma, as well as the insights gained throughout the work.

Índice

1 Introducción

En los últimos años, ha vuelto con fuerza una pregunta que ronda desde hace décadas: ¿pueden las máquinas hacer matemáticas?

La respuesta, en general, sigue siendo no. Pero con los avances recientes en inteligencia artificial y verificación formal, esta barrera empieza a tambalearse. Algunos sistemas ya han conseguido demostrar de manera automatizada resultados del nivel de las olimpiadas matemáticas [1].

Un primer desafío a la hora de intentar que un ordenador haga matemáticas es cerrar la brecha entre el lenguaje que nosotros entendemos y el que una máquina puede procesar. A pesar de que el lenguaje matemático es ya bastante preciso, todavía existe un desfase considerable entre una demostración escrita en un artículo y una que un ordenador pueda validar.

Algunos primeros intentos de crear un lenguaje en el que las matemáticas fueran comprensibles para una máquina se remotan a proyectos como *Automath* (1967) y *Mizar* (1973). A día de hoy, herramientas como Coq, Isabelle, HOL o Lean permiten escribir y verificar matemáticas con asistencia del ordenador. A este tipo de herramientas se las conoce como verificadores de demostraciones interactivos o ITP (*Interactive Theorem Provers*) [2].

En este trabajo me centro en el aprendizaje y uso de Lean, una de estas herramientas. Me interesa especialmente esta intersección entre matemáticas y computación: el proceso de escribir matemáticas de forma que un ordenador las acepte como válidas. Para mí, formalizar demostraciones en Lean es parecido a resolver un puzzle complejo y desafiante.

He elegido Lean por encima de otros verificadores porque tenía a mi disposición varios recursos para facilitar su aprendizaje, como cursos impartidos en mi Facultad y una gran cantidad de documentación online, gracias a la comunidad de Lean. Esta comunidad es muy activa con miles de usuarios y una extensa librería de resultados matemáticos formalizados listos para su reutilización. También valoro que su sintaxis sea más cercana al lenguaje matemático habitual que la de otros sistemas.

En particular, he decidido trabajar sobre resultados de topología porque me parece una rama especialmente interesante (probablemente influida por la fama de las tazas y los donuts) y porque es un área suficientemente abstracta como para poner a prueba las capacidades del lenguaje formal, además de las mías propias, en este puzzle de piezas topológicas.

1.1 Objetivos

El objetivo principal es adquirir un conocimiento sólido del asistente de demostración Lean, llegando a ser capaz de formalizar resultados sencillos del grado en Matemáticas.

Una vez alcanzada cierta familiaridad con el sistema, se plantea como objetivo adicional la formalización de un resultado relevante que permita ilustrar las dificultades del proceso de demostrar resultados en un lenguaje formal.

Para ello, se ha elegido el lema de Urysohn, un resultado central de la topología general cuya demostración requiere, entre otras herramientas, el uso de distintos tipos de inducción. Esto permite alcanzar un manejo más avanzado del sistema de Lean, al involucrar características del mismo que no son elementales.

Finalmente, mediante este proceso de aprendizaje y puesta a prueba se quiere llegar a comprender las ventajas y las dificultades que supone trasladar las matemáticas al entorno de un asistente de demostración.

1.2 Plan de trabajo

El primer paso fue adquirir los conocimientos básicos sobre Lean. Para ello, me inscribí al curso de Lean impartido por CompuMates en la Facultad de Matemáticas durante el curso 2023/24. Esto me permitió tener unas primeras nociones básicas y empezar a escribir pequeñas demostraciones en Lean 3, centradas principalmente lógica proposicional.

A partir de entonces, mi interés por Lean creció, y comencé a estudiar de manera autodidacta la versión Lean 4, más reciente y actualmente en desarrollo activo. En particular, seguí con detalle el curso en línea *Formalising Mathematics 2024* impartido por Kevin Buzzard [3], completando los ejercicios propuestos hasta el capítulo 10, dedicado a los espacios topológicos.

Dicho capítulo era muy escueto, e incluía únicamente dos ejemplos de espacios topológicos concretos y algunas cuestiones sobre continuidad de funciones. Los ejercicios que resolví en esta etapa inicial pueden consultarse en el repositorio `https://github.com/pepamontero/Lean4-Buzzard-Exercises`.

Ante la escasez de contenidos sobre topología, decidí reforzar este aprendizaje tomando como referencia los apuntes que había tomado durante el curso de Topología Elemental del grado. Comencé a formalizar los principales resultados que había visto en clase. Parte de este trabajo se presenta en la sección 3 de este documento.

Al avanzar en esta tarea, llegué hasta el tema de los axiomas de separación. Como durante el curso no habíamos trabajado con espacios normales, recurrí

a otras fuentes para comprender esta noción. Fue entonces cuando formulé por primera vez en Lean el enunciado del lema de Urysohn y, a partir de ahí, enfoqué el resto del trabajo en su formalización.

Finalmente, dado que este resultado ya estaba demostrado en la biblioteca oficial Mathlib, he consultado dicha implementación con el objetivo de comprender su enfoque y comparar las ideas fundamentales con las utilizadas en mi propia formalización.

1.3 Estructura del trabajo

Este trabajo se organiza en cuatro partes principales. En primer lugar, se presenta una introducción detallada al lenguaje Lean, incluyendo sus fundamentos matemáticos y su uso como asistente de demostración matemática.

A continuación, se introducen algunos resultados de topología general. El objetivo de esta sección no es desarrollar una teoría completa, sino ilustrar mediante ejemplos concretos cómo se formalizan propiedades topológicas y cómo se describen espacios en Lean.

La tercera parte constituye el núcleo del trabajo: se expone de manera detallada el proceso de formalización del lema de Urysohn y su demostración. Se detalla el camino a seguir junto con las principales dificultades encontradas durante el proceso y las estrategias seguidas para resolverlas. La sección concluye con la comparación con las ideas utilizadas en la demostración de Mathlib.

Todo el código desarrollado durante el proyecto, incluyendo las definiciones y demostraciones en Lean, se encuentra disponible en este repositorio público: `https://github.com/pepamontero/tfg-topologia-lean4`. Este repositorio complementa la memoria escrita y permite reproducir o consultar fácilmente los resultados formales obtenidos.

Por último, se presentan las conclusiones, centradas en las dificultades y ventajas que plantea la escritura de demostraciones en un entorno formal como Lean.

2 Lean Theorem Prover

A medida que las matemáticas se vuelven más técnicas y especializadas, verificar con rigor las demostraciones formales es una tarea cada vez más costosa. Con la motivación de facilitarla, en las últimas décadas ha surgido un interés por la verificación computacional de teoremas, dando lugar al desarrollo de sistemas como Lean, Coq o Isabelle.

Dentro de este campo, distinguimos dos tipos de sistemas de verificación formal: interactivos (ITP), que proporcionan un entorno en el que el usuario guía el proceso de la demostración paso a paso, centrándose en el aspecto de "verificación", y automáticos (ATP), que buscan completar demostraciones de manera completamente autónoma [4, Sección 1].

En este trabajo nos centraremos en el uso de **Lean Theorem Prover**, introducido en 2013 por Leonardo de Moura desde Microsoft Research. Se trata de un verificador cuyo objetivo es reducir la distancia entre demostraciones asistidas y automatizadas, combinando un lenguaje basado en la teoría de tipos dependientes con herramientas que permiten delegar sub-problemas sencillos al sistema

Aunque aquí nos limitaremos a su uso como asistente de demostración, Lean es también un lenguaje de programación funcional completo, lo que ofrece amplias posibilidades de personalización y automatización al usuario [4, Sección 1].

En este sistema, es posible definir objetos matemáticos, especificar propiedades sobre ellos y demostrar que dichas propiedades se cumplen. Esta tarea se ve facilitada por *Mathlib*, una extensa biblioteca de matemáticas formalizadas en Lean desarrollada de manera colaborativa por una comunidad activa y en constante crecimiento [5].

Las demostraciones son verificadas automáticamente por el núcleo lógico de Lean, que garantiza su corrección mediante un sistema de tipos expresivo y riguroso. La fiabilidad de Lean como asistente de demostración reside precisamente en la simplicidad y robustez de este núcleo [6].

En esta sección seguiremos principalmente el manual en línea *Theorem Proving in Lean 4* [4] que es una versión actualizada del libro *Theorem Proving in Lean* [7] publicado en 2021 para adaptarse a la nueva versión de Lean. A nivel teórico, no existe una gran diferencia entre los dos, por lo que ambas referencias son válidas para comprender los fundamentos que exponemos aquí.

2.1 La teoría de tipos de Lean

La teoría de conjuntos de Zermelo-Fraenkel con el axioma de elección (ZFC) es la base fundacional elegida para formalizar la mayoría de las matemáticas que

conocemos. En este marco, todos los objetos matemáticos (números, funciones, estructuras algebraicas, etc.) pueden representarse como conjuntos, construidos a partir de unos pocos axiomas básicos.

Sin embargo, este sistema carece de una estructura interna diferenciada: todo objeto matemático, como un número, una función o incluso una colección de funciones son, en última instancia, conjuntos. Para lograr una representación más clara y diferenciada de los objetos matemáticos, Lean utiliza, en su lugar, un sistema basado en tipos. Además, este enfoque nos ofrece la posibilidad de establecer una correspondencia entre programas y demostraciones matemáticas, conocida como la correspondencia de Curry-Howard[1].

En particular, Lean se fundamenta en el *Cálculo de Construcciones Inductivas*, una extensión del cálculo de tipos dependientes que incorpora tipos inductivos y una jerarquía numerable no acumulativa de universos [9]. Aunque no es necesario entender este sistema para utilizar Lean como asistente de demostración, a continuación daremos una breve explicación de los conceptos fundamentales: la teoría de tipos, el cálculo lambda, la incorporación de tipos a esta última, y la introducción de tipos dependientes.

En esta sección, veremos varios fragmentos de código en Lean. Lean cuenta con un compilador interactivo que procesa cada línea cuando el cursor se encuentra sobre ella, mostrando el resultado por pantalla. A partir de ahora, los comentarios que acompañan al código reflejan la salida que Lean devuelve en cada línea. Los comentarios en Lean se escriben empezando con doble guión ($--$) y están en color gris.

2.1.1 Teoría de tipos

Empecemos por lo más básico: la teoría de tipos. Cambiamos el paradigma de "cada objeto es un conjunto", propio de ZFC, a "cada objeto es un término con un tipo asociado". Esto nos permite estructurar con mayor claridad los objetos matemáticos y sus relaciones.

Por ejemplo, 3 es un término de tipo "natural" ($\mathbb{N}$), mientras que "true" es un término de tipo "booleano". En Lean, podemos comprobar el tipo de estas expresiones utilizando el comando `#check`.

```
#check 3    -- 3 : ℕ
#check true   -- Bool.true : Bool
```

[1] La correspondencia de Curry-Howard establece una relación entre lógica y programación; permite entender como pueden ser equivalentes "demostrar una proposición" y "construir un término de cierto tipo". Veremos qué quiere decir esto en la práctica más adelante, pero las ideas más profundas, que quedan fuera del alcance de este trabajo, se exponen detalladamente en [8].

Como en este ejemplo, en Lean utilizamos el símbolo : para describir la información sobre el tipado. Es decir, si x es un término de tipo X, escribimos $x : X$.

Por otro lado, un tipo, como es $\mathbb{N}$, también es un término. Podemos comprobar su tipo:

```
#check ℕ    -- ℕ : Type
```

En Lean, los tipos tienen su propio tipo, que recibe el nombre de `Type`. Esto nos permite definir nuevos tipos. Podemos utilizar el comando `variable` para definir objetos en nuestro código[2].

```
variable (X : Type)
#check X    -- X : Type
variable (x : X)
#check x    -- x : X
```

Ahora, podemos combinar distintos tipos para obtener tipos más complejos. Sean X e Y dos tipos. Podemos considerar el tipo $X \times Y$, que denota los pares formados por un elemento de X y otro de Y. El tipo que más utilizaremos es $X \to Y$, que denota las funciones de X en Y. Escribimos esto en Lean.

```
variable (X Y : Type)
#check X × Y    -- X ×  Y :Type
variable (x : X) (y : Y)
#check (x, y)    -- (x, y) : X × Y

#check X → Y    -- X → Y : Type
variable (f : X → Y)
#check f    -- f : X → Y
```

Por otro lado, a partir de la yuxtaposición de términos simples, podemos formar términos más complejos. En Lean, las reglas de tipado dictan el tipo de estos nuevos términos obtenidos. Por ejemplo, si x es de tipo X y f es de tipo $A \to B$, como en el ejemplo anterior, entonces fx tiene tipo B. En efecto:

```
#check f x    -- f x : Y
```

Como ocurre en el ejemplo anterior, Lean puede deducir automáticamente el tipo de muchos términos a partir del contexto. Esta capacidad se conoce como **inferencia de tipos** y en muchas ocasiones nos facilita el trabajo, permitiendo un código más conciso.

[2]Veremos este comando en detalle más adelante.

2.2 El cálculo lambda

El Cálculo Lambda, introducido por Alonzo Church en los años 1930, es un sistema formal que permite construir expresiones mediante dos operaciones básicas: la abstracción y la aplicación [10]. En este sistema, algunas expresiones representan funciones (usando la notación lambda) y otras representan valores sobre los que pueden aplicarse dichas funciones. Dos expresiones pueden yuxtaponerse para formar una nueva expresión; si la primera es una abstracción, entonces se interpreta como aplicación funcional. A partir de ahí, pueden realizarse reducciones para simplificar la expresión resultante si corresponde.

Por ejemplo, un término válido podría ser $\lambda n, n+2$, que representa una función que puede aplicarse a un valor para obtener otro.

En Lean, definimos funciones utilizando el comando `fun`, que corresponde a la notación λ del cálculo lambda clásico[3]. Por ejemplo[4]:

```
def f := fun n ↦ n + 2
```

Además, Lean admite reducción por aplicación funcional sobre estos términos. Por ejemplo, si aplicamos la anterior función a 3, $(n \mapsto n + 2)3$, se puede reducir a $3+2$ por aplicación funcional, y, suponiendo que la operación $+$ estaba definida anteriormente, podemos reducir esta expresión a 5. Podemos comprobar el resultado de esta reducción utilizando el comando `#eval`.

```
#eval f 3    -- 5
```

Diremos que dos términos que pueden reducirse de esta manera al mismo valor son **iguales por definición**. Lean trata términos que sean iguales por definición como literalmente iguales, como veremos en la práctica.

2.2.1 Cálculo lambda tipado

El cálculo lambda clásico no incorpora tipos: cualquier función puede aplicarse a cualquier argumento. Para evitar inconsistencias y dar más estructura, se introduce el cálculo lambda tipado, donde a cada término se le asocia un tipo concreto.

Podemos escribir el ejemplo anterior de la siguiente forma en Lean:

```
def f : ℕ → ℕ := fun n => 2 * n
```

[3] En la versión anterior de Lean, se utilizaba la notación `λ n, n + 2`, sin embargo en esta última versión se ha cambiado a `fun n ↦ n + 2` para mejorar la legibilidad del código.

[4] Estudiaremos el comando `def` en detalle más adelante.

Aquí, estamos indicando que f tiene tipo $\mathbb{N} \to \mathbb{N}$, es decir, que es una función que toma valores en $\mathbb{N}$ y devuelve valores en $\mathbb{N}$. Esta información permite a Lean verificar que las expresiones están bien formadas.

2.2.2 Teoría de tipos dependientes

Para poder expresar los distintos objetos matemáticos en esta teoría, necesitamos introducir los tipos dependientes. Un tipo dependiente es aquel que puede variar en función de un término.

Por ejemplo, en Lean, el tipo `List α` representa una lista de elementos de tipo α. Si definimos un objeto de tipo `List α`, el tipo de este objeto dependerá del tipo α que le asignemos Internamente, se define como una función de tipo `Type u → Type u`.

```
#check List    -- List.{u} (α : Type u) : Type u
#check List ℝ    -- List ℝ : Type
```

Podemos pensar en los tipos dependientes como una generalización de las funciones que van de un tipo en otro tipo, y el tipo de llegada puede depender del tipo de entrada. Llamamos "funciones dependientes" a este tipo de funciones.

En particular, la proposición $\forall x \in \mathbb{N}, Px$ se representa en Lean mediante un tipo dependiente que expresa el conjunto de las funciones que, dado un elemento $x \in \mathbb{N}$, devuelve una prueba de Px. La notación de Lean para este tipo de construcción es `Π x : ℕ, P x`. El operador Π sirve para denotar que el tipo de salida puede depender del valor de entrada.

2.2.3 Jerarquía de universos

Puesto que en la teoría de tipos cada elemento tiene un tipo, también el tipo `Type` tiene un tipo asociado. Pero si escribiésemos simplemente `Type : Type`, el sistema caería en una inconsistencia similar a la paradoja de Russel. Para evitar esto, Lean introduce una jerarquía infinita de universos: `Type 0 : Type 1`, `Type 1 : Type 2`, y así sucesivamente.

Esta jerarquía no es acumulativa, lo que significa que si `A : Type u`, no se asume en general que `A : Type (u+1)`. Esto permite que Lean controle mejor cómo se combinan los tipos y evite ambigüedades al determinar a qué universo pertenece cada término, aunque puede realizar ciertas conversiones automáticamente cuando es seguro hacerlo. Por ello, en la mayoría de los casos no es necesario trabajar explícitamente con estos universos.

2.2.4 Tipos inductivos

En Lean, la gran mayoría de tipos son instancias de una familia de tipos conocidos como **tipos inductivos**. Un tipo inductivo es una estructura formada por una lista finita de constructores, cada uno con su tipo correspondiente. Cada constructor describe una forma válida de construir un término de este nuevo tipo.

En Lean, definimos un tipo inductivo utilizando la palabra clave `inductive`[5].

```
inductive Foo where
  | constructor₁ : ... → Foo
  | constructor₂ : ... → Foo
  ...
  | constructor_n : ... → Foo
```

Un ejemplo clásico de definición inductiva es el conjunto de los números naturales, $\mathbb{N}$. En Lean, podemos describir el tipo `Nat` de los números naturales como

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

Internamente, la declaración `inductive` genera automáticamente una colección de axiomas que definen el tipo:

- Una constante, `Nat`, que representa el nuevo tipo.
- Una serie de reglas de introducción o constructores, que indican las posibles formas de construir términos del nuevo tipo.
- Una regla de eliminación, `Nat.rec`, que indica la forma de "usar" un término de este tipo[6].

```
#print Nat.rec
    -- recursor Nat.rec.{u}  :  {motive : ℕ → Sort u} → motive
  Nat.zero → ((n : ℕ) → motive n → motive n.succ) → (t : ℕ) →
  motive t
```

Es decir, `inductive` puede verse como *azúcar sintáctico* que genera automáticamente el siguiente código en Lean[7]:

[5] Aunque en Lean los tipos inductivos se introducen como una construcción primitiva del lenguaje, pueden definirse de manera equivalente sólo en términos de tipos dependientes. Esta reducción se explora formalmente en [11].

[6] El comando `#print` muestra la definición completa del objeto, a diferencia de `#check`, que solo muestra su tipo.

[7] Estudiaremos el comando `axiom` en detalle más adelante.

```
axiom (Nat : Type)
axiom (zero : Nat)
axiom (succ : Nat → Nat)
axiom (Nat.rec : {motive : Nat → Sort u} → motive Nat.zero →
  ((n : Nat) → motive n → motive Nat.succ n) → (t : Nat) →
  motive t)
```

Este último objeto, `Nat.rec`, codifica el principio de inducción sobre los naturales[8]. Este principio se utiliza implícitamente en muchas definiciones por casos, como por ejemplo:

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

En esta definición, utilizamos la expresión `match n with` para distinguir los dos posibles casos de un número natural: `zero` y `succ n`. Internamente, Lean compila esta expresión como una aplicación de `Nat.rec`. Veremos más adelante cómo este principio de inducción puede utilizarse no solo para definir funciones, sino también para probar propiedades sobre todos los términos de un tipo inductivo.

Cabe destacar que existen otras construcciones en Lean, como `structure` o `class`, que se definen internamente como casos particulares de tipos inductivos, pero se añaden como construcciones separadas para añadir legibilidad y funcionalidad. Veremos algunos ejemplos de su uso a lo largo del trabajo.

Finalmente, mediante los tipos inductivos es posible definir los conectores lógicos (negación, conjunción, disyunción e implicación). Esto constituye otra gran diferencia entre la teoría de conjuntos y el cálculo de construcciones inductivas. Para utilizar la teoría de conjuntos, es necesario haber desarrollado previamente la lógica (de primer orden). De esta manera, las demostraciones formales no constituyen objetos matemáticos, sino que viven exclusivamente en el plano meta-teórico.

En el cálculo de construcciones inductivas, en cambio, la lógica se expresa dentro de la misma teoría, y las demostraciones son objetos matemáticos que viven dentro de ella.

[8] `Nat.rec` es un tipo que depende de `motive`, que es una propiedad cualquiera sobre los naturales. `Nat.rec` nos dice que si se cumple `motive` para `Nat.zero` (`motive Nat.zero`), entonces si para cada `n` (`n : Nat`) que cumpla `motive` (`motive n`) se tiene que `n+1` cumple `motive` (`motive Nat.succ n`), entonces se cumple `motive` para cualquier `n` (`(t : Nat) → motive t`).

2.2.5 Las demostraciones como objeto matemático

Las proposiciones, como cualquier otro objeto en esta teoría, son términos con un tipo asociado. En Lean, este tipo recibe el nombre de `Prop`.

```
#check Prop    -- Prop : Type
#print True    -- inductive True : Prop
```

```
variable (P : Prop)
#check P    -- P : Prop
#check ¬ P    -- ¬ P : Prop
```

En Lean, interpretamos los objetos de tipo `Prop` como tipos en sí mismos y las demostraciones de cada proposición como términos que habitan este tipo, siguiendo la correspondencia de Curry-Howard. Es decir, una proposición `p : Prop` es el tipo de las demostraciones de `p`; una expresión de la forma `h : p` quiere decir que `h` es una demostración de `p`. Decimos que una proposición `p` es verdadera si podemos construir término de tipo `p`.

```
variable (p : Prop)
variable (h : p)
#check h    -- h : p
```

Esto, junto con la teoría de tipos dependientes, nos proporciona una forma de definir cualquier resultado matemático. Por ejemplo, "ser par" es una propiedad que depende de un número natural n, por lo que podríamos describirlo mediante `es_par : ℕ → Prop`. Para cada `n` natural, obtenemos un término de tipo `Prop`.

```
def es_par : ℕ → Prop := ...
#check es_par    -- es_par : ℕ → Prop
#check es_par 3    -- es_par 3 : Prop
```

En este caso, un término de tipo `es_par n` será una prueba de que `n` es par.

Además, si `p : Prop` es una proposición, Lean reconoce cualesquiera dos elementos de tipo `p` (`h1 h2 : p`) como iguales por definición: no importa qué prueba concreta tengamos, sólo importa su existencia. Esto se conoce como "irrelevacia de las demostraciones" (*proof irrelevance*).

Esta propiedad tiene consecuencias importantes. Por un lado, evita comportamientos no deseados cuando definimos estructuras que dependen de proposiciones. Por ejemplo, si quisiéramos definir un punto del primer cuadrante en $\mathbb{R}^2$ como un par de la forma `x y : ℝ × ℝ` junto con una demostración `h : x ≥ 0 ∧ y ≥ 0`, entonces, gracias a la irrelevancia de las demostraciones, podemos identificar dos puntos que tengan las mismas coordenadas, porque las pruebas `h` y `h'` asociadas a cada uno son la misma.

Por otro lado, esta misma propiedad impide acceder al contenido de una prueba. En particular, no es posible extraer directamente el testigo de la demostración de una proposición existencial del tipo $\exists x, P(x)$, ya que todas las demostraciones de esa proposición se consideran iguales. Dada una demostración de $\exists x, P(x)$, contamos varios métodos para obtener un testigo, uno de los cuales es usar el axioma de elección. Volveremos sobre esta cuestión más adelante.

Una última característica destacable de las proposiciones en Lean es que la implicación lógica se representa directamente mediante funciones: dadas dos proposiciones `p q : Prop`, una prueba de `p → q` es simplemente una función que, dada una prueba de `p`, devuelve una prueba de `q`. Esta identificación entre funciones e implicaciones es otra manifestación de la correspondencia de Curry-Howard.

En resumen, para poder expresar un resultado matemático en este lenguaje, tenemos que escribir un término de la forma `p : Prop`. Para probar que el resultado es cierto, debemos construir un término `h : p`. El trabajo de Lean como asistente de demostración es verificar que el término `h` está bien construido y tiene el tipo correcto.

2.3 ¿Por qué fiarnos de Lean?

Ahora que hemos descrito la manera en la que un resultado se considera demostrado en Lean, tiene sentido hacerse la pregunta: ¿por qué deberíamos fiarnos de la inferencia de tipos de Lean? ¿Qué garantías tenemos de que las demostraciones que Lean acepta, son realmente correctas?

Como hemos señalado, demostrar un resultado en Lean consiste en construir correctamente un término que tiene un determinado tipo. Este proceso es análogo al de verificar programas: se trata de comprobar que un término está bien formado (siguiendo unas reglas concretas) y satisface una especificación dada, expresada como un tipo. Esta tarea recae sobre el núcleo (o *kernel*) de Lean, un pequeño programa que contiene la implementación mínima de la lógica interna de Lean.

El resto de componentes de Lean con el que interactuamos para construir demostraciones (como por ejemplo las tácticas que veremos después) devuelven construcciones expresadas en el lenguaje del kernel de Lean [6]. Esto quiere decir que confiar en Lean realmente se reduce a confiar en su kernel[9].

Ahora, ¿por qué nos fiamos del kernel de Lean? Gracias a que el kernel es pequeño y está aislado del resto del sistema, es posible escribir implementaciones independientes del mismo que verifiquen de manera autónoma las demostracio-

[9]Esta idea se conoce como *criterio de de Bruijn*, que propone que un verificador formal debe producir sus pruebas en el lenguaje de un núcleo pequeño, incluso aunque utilicen otros métodos más complicados para construir dichas pruebas a priori [6].

nes aceptadas por Lean. Lean permite exportar estas demostraciones en un formato intermedio que contiene toda la información necesaria para reconstruirlas y validarlas externamente. Además, puesto que este formato modular, es posible validar solo ciertos aspectos concretos del kernel [6]. Por ejemplo, en [12], Carneiro describe una nueva implementación externa del verificador de tipos de Lean 4, escrita en el propio lenguaje Lean y capaz de verificar toda la biblioteca de Mathlib.

2.4 Demostraciones en Lean

Hasta ahora, hemos explorado la teoría de tipos dependientes sobre la que se construye Lean, así como los fundamentos que garantizan la corrección de sus demostraciones. Pasamos por tanto a un enfoque más práctico: ¿cómo escribimos matemáticas en Lean?

Recordemos que formalizar un resultado en Lean no consiste solo en escribir su enunciado, sino también en construir una demostración paso a paso, sin omisiones y con total precisión. Aquí, nunca nos basta con escribir "trivial" cuando creamos que algo ya deberíamos poder saberlo: necesitamos convencer al sistema de que cada paso es válido.

Esta sección está dedicada a aprender a escribir demostraciones en Lean. Veremos cómo introducir nuevos objetos en nuestro contexto, cómo enunciar proposiciones y cómo construir demostraciones interactuando con Lean. También presentaremos algunas herramientas de automatización y métodos para poder apoyarnos en la librería de Mathlib.

2.4.1 Axiomas, definiciones y variables

Antes de escribir demostraciones en cualquier sistema formal, necesitamos describir el **contexto** en el que trabajamos: el conjunto de objetos e hipótesis disponibles en un momento dado. Este contexto es dinámico y se va ampliando a medida que introducimos nuevos elementos.

En Lean ocurre lo mismo. El sistema mantiene y actualiza este contexto constantemente para comprobar que cada expresión está bien formada y tiene el tipo esperado.

Podemos introducir nueva información en el contexto de distintas formas. Distinguimos entre axiomas, definiciones y variables, cada una con una función lógica distinta en el sistema.

- **Axiomas**[10]

[10] En Lean 3, a este tipo de declaraciones se les llamaba *constantes* y utilizaban el comando

Permiten introducir hipótesis que se asumen sin demostración. En particular, escribir que x "es de tipo X" es también una hipótesis, por lo que los axiomas pueden utilizarse para introducir nuevos objetos[11]. Por ejemplo:

```
axiom P : Prop
axiom h : P → P
```

Estamos declarando una proposición P y una prueba de que P implica P.

```
axiom n : ℕ
axiom hn : n > 2
```

Aquí estamos suponiendo que n es un número natural mayor que 2.

Así, los axiomas nos permiten fijar hechos que queremos asumir como válidos a lo largo de nuestras demostraciones.

- **Definiciones**

Introducen objetos nuevos a partir de otros ya conocidos. A diferencia de los axiomas, no basta con indicar el tipo del nuevo objeto, sino que también hay que dar su construcción. Por ejemplo:

```
def f : ℕ → ℕ := fun n ↦ 2 * n
def n : ℕ := 3
def es_par : ℕ → Prop := fun n ↦ ∃ m, n = f m
```

Además, cuando el tipo puede inferirse a partir de la construcción, no es necesario indicarlo explícitamente:

```
def n := 3
#check n    -- n : ℕ
```

- **Variables**

En la mayoría de lenguajes de programación, estamos acostumbrados a que definir una variable implique asignarle un valor concreto. Sin embargo, en Lean las variables se comportan más bien como en lógica. Al introducir una variable x, lo que se introduce es un contexto universal: siempre que x aparezca de

`constant`.

[11] En este sentido decíamos que definir un tipo inductivo es análogo a escribir una colección de axiomas. `inductive Nat` se puede ver como una versión estructurada de `axiom Nat : Type`, `axiom zero : Nat`, `axiom succ : Nat to Nat`, etc.

forma libre, Lean interpretará que lo que sigue está cuantificado universalmente respecto a x. Por ejemplo[12]:

```
variable (x : ℕ)
axiom hx : x ≥ 0
#print hx    -- axiom hx : ∀ (x : ℕ), x ≥ 0
```

2.4.2 Proposiciones

Además de introducir objetos, también queremos enunciar y demostrar proposiciones. En Lean, esto se hace del mismo modo que en otros sistemas formales: primero escribimos los resultados (como lemas o teoremas) formalmente, y después proporcionamos una demostración.

Como ya hemos visto, una proposición en Lean es un término de tipo `Prop`, y una demostración de `p : Prop` es simplemente un término de tipo `p`. Por tanto, demostrar una proposición no es diferente de definir un objeto; podemos utilizar `def` para escribir resultados matemáticos. Por ejemplo:

```
def mi_prop : 1 > 0 := ...
```

Aquí estamos diciendo que `mi_prop` es un objeto de tipo `1 > 0`. Si en el lugar de `...` proporcionamos un término de tipo `1 > 0`, habremos demostrado `mi_prop`.

Sin embargo, para mayor claridad y estructura, Lean proporciona los comandos `lemma` y `theorem`. Ambos funcionan exactamente igual que `def` y son intercambiables entre sí, pero facilitan la lectura del código indicando qué objetos son resultados matemáticos, y la jerarquía de importancia entre ellos. La sintaxis es la misma:

```
lemma my_lemma : 1 > 0 := ...
theorem commutative_sum (a b : ℕ) : a + b = b + a := ...
```

Existe también el comando `example`, que sirve para escribir demostraciones sin la necesidad de nombrar el resultado:

```
example (a b : ℕ) : a * b = b * a := ...
```

Este tipo de expresiones no amplían el contexto ni definen nuevos objetos; son simplemente comprobaciones locales. Pero veremos que nos pueden ser útiles en ciertas ocasiones.

[12]Las variables, a diferencia de los axiomas y las definiciones, se escriben entre paréntesis. Lo mismo ocurre con los argumentos que toman las proposiciones, como veremos más adelante. Esto está relacionado con la correspondencia de Curry–Howard: declarar una variable equivale a abstraer sobre ella, lo que corresponde a cuantificar universalmente.

2.4.3 Demostraciones: el modo táctico

Llegamos a la parte central de esta sección: **escribir demostraciones** en Lean. En general, hay dos formas de construir una demostración en Lean:

- Mediante **términos**, es decir, escribiendo directamente una expresión del tipo deseado.
- Mediante el **modo táctico**, en el que una demostración se construye paso a paso usando instrucciones llamadas **tácticas**.

En este trabajo utilizaremos exclusivamente el modo táctico, ya que es el enfoque más práctico y más cercano a la forma en que razonamos al escribir demostraciones matemáticas en lenguaje natural.

En una demostración informal, solemos avanzar mediante pasos lógicos encadenados: "supongamos que...", "entonces...", "por el lema..., se tiene...". Cada uno de estos pasos se traduce en Lean mediante una táctica: una instrucción que modifica el estado de la demostración, ya sea introduciendo hipótesis, aplicando resultados conocidos, dividiendo el objetivo en partes más manejables, etc.

Además, el modo táctico nos permite trabajar de manera **interactiva** con Lean. Si escribimos un enunciado, e inmediatamente después de `:=` escribimos `by`, estamos indicando a Lean que para la construcción de este término vamos a utilizar el modo táctico. Por ejemplo:

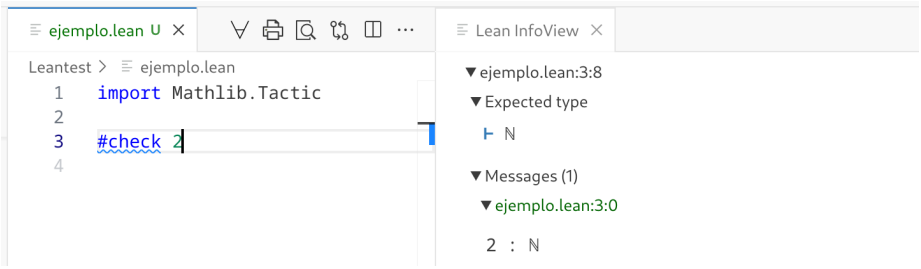
```
example (p q : Prop) (hp : p) (hq : q) : p ∧ q := by
```

Estamos indicando que queremos construir un término de tipo `p ∧ q` a partir de las hipótesis `hp : p` y `hq : q`, y que para esa construcción vamos a utilizar el modo táctico.

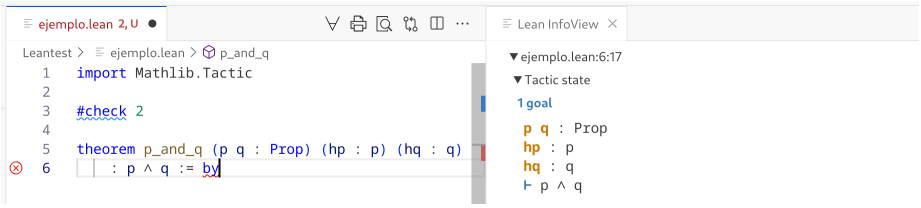
Internamente, Lean interpreta esto mediante la generación de un contexto local (nuestras hipótesis) y un **objetivo** (nuestra tesis), que consiste en construir el término del tipo esperado. Después de `by`, podemos empezar a escribir tácticas, que Lean interpretará actualizando el contexto y el objetivos.

Este objetivo aparece reflejado en el **InfoView**, una ventana que muestra el estado actual de nuestra demostración. Lean procesa línea a línea de forma automática, por lo que en cualquier momento podemos consultar el impacto de haber aplicado una táctica simplemente colocando el cursor sobre la línea de código correspondiente.

De hecho, en el InfoView también se muestran los resultados de las instrucciones que ya hemos visto como `#check`, `#print` o `#eval`.



En general, mientras escribimos en Lean, tendremos abierta esta ventana paralelamente a nuestro código, para poder ir viendo el progreso de nuestra demostración.

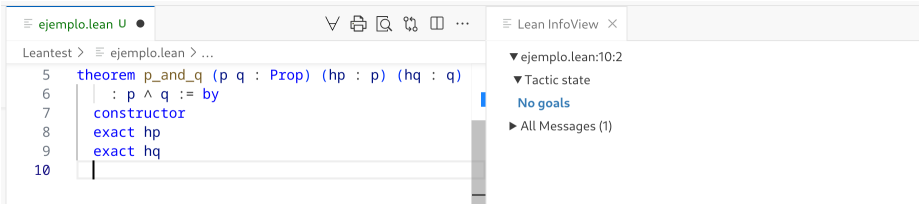


Bajo el apartado `Tactic state`, podemos comprobar:

- El número de tesis que nos quedan por demostrar (en este caso solo una: `1 goal`).
- Nuestro contexto.
- La (o las) tesis, marcada con el símbolo `⊢`.

A partir de este punto, podemos empezar a añadir las tácticas que van a constituir nuestra demostración. Las tácticas se escriben una detrás de otra, separadas por punto y coma (`;`) o por saltos de línea.

Al escribir una táctica, el apartado `Tactic state` del InfoView se actualizará según corresponda. Cuando todas las tesis se hayan resuelto, el InfoView mostrará `No goals`.



2.4.4 Algunas tácticas básicas

En esta sección veremos algunas de las tácticas más básicas y útiles para construir demostraciones en Lean. Veremos cómo se aplican y qué efecto tienen en el InfoView. El resto de tácticas que aparecen a lo largo del trabajo pueden consultarse en el apartado de tácticas de la documentación de Lean, [13].

Para poder utilizar las tácticas mencionadas a continuación, es necesario importar el módulo de Mathlib correspondiente al modo táctico mediante

```
import Mathlib.Tactic
```

A partir de aquí, en lugar de mostrar capturas del InfoView, utilizaremos dos bloques de código en paralelo: el de la izquierda contiene el código de Lean; el de la derecha representa el estado que se mostraría en el InfoView si colocásemos el cursor en la última línea.

`intro`

La táctica `intro` introduce un nuevo objeto en el contexto, de manera similar a escribir "Supongamos que..." o "Sea..." en una demostración informal.

Es útil cuando el objetivo tiene la forma de una implicación o un cuantificador universal: transformamos la primera parte de la tesis en una nueva hipótesis y la segunda en la nueva tesis. Por ejemplo, para la implicación:

```
example (p : Prop) : p → p := by
```

```
Tactic state
  1 goal
  p : Prop
  ⊢ p → p
```

↓

```
example (p : Prop) : p → p := by
  intro hp
```

```
Tactic state
  1 goal
  p : Prop
  hp : p
  ⊢ p
```

Y para deshacer cuantificadores:

```
example : ∀ (p : Prop), p → p := by
```

```
Tactic state
  1 goal
  ⊢ ∀ (p : Prop), p → p
```

↓

```
example : ∀ (p : Prop), p → p := by
  intro q
```

```
Tactic state
  1 goal
  q : Prop
  ⊢ q → q
```

`exact`

La táctica `exact` se utiliza cuando ya tenemos, en nuestro contexto, exactamente lo que queremos demostrar. Es decir, existe una hipótesis que coincide con la tesis actual. Por ejemplo:

```
example : ∀ (p : Prop), p → p := by
  intro p hp
```

```
Tactic state
  1 goal
  p : Prop
  hp : p
  ⊢ p
```

↓

```
example : ∀ (p : Prop), p → p := by
  intro p hp
  exact hp
```

```
Tactic state
  No goals
```

A modo de comparación, podríamos construir esta demostración utilizando sólo términos de la siguiente forma:

```
example : ∀ (p : Prop), p → p := fun p ↦ (fun hp : p ↦ hp)
```

En este caso puede resultar más sencillo, pero en demostraciones más complejas perdemos legibilidad.

`apply`

La táctica `apply` nos permite usar una implicación para reducir un objetivo a otro más simple. Equivale a utilizar la regla del *Modus Ponens*: si tenemos una hipótesis de la forma $p \to q$ y queremos demostrar q, basta con demostrar p.

```
example : (p q : Prop) (hp : p)
    (hpq : p → q) : q := by
```

```
Tactic state
  1 goal
  p q : Prop
  hp : p
  hpq : p → q
  ⊢ q
```

↓

```
example : (p q : Prop) (hp : p)
    (hpq : p → q) : q := by
  apply hpq
```

```
Tactic state
  1 goal
  p q : Prop
  hp : p
  hpq : p → q
  ⊢ p
```

Podríamos completar esta demostración usando `exact hp`.

`use`

Utilizamos `use` para trabajar con el cuantificador existencial. Si queremos demostrar una proposición de la forma "$\exists x, Px$", basta con encontrar un x_0 concreto que satisfaga la propiedad P.

En este caso, aplicamos `use` para indicarle a Lean el valor concreto x_0 que queremos usar para demostrar la existencia. El objetivo pasa a ser entonces demostrar que x_0 satisface P. Por ejemplo:

```
example : ∃ n : ℕ, n > 3 := by
```

```
Tactic state
  1 goal
  ⊢ ∃ n, n > 3
```

↓

```
example : ∃ n : ℕ, n > 3 := by
  use 5
```

```
Tactic state
  1 goal
  ⊢ 5 > 3
```

`left` , `right`

Las tácticas `left` y `right` se utilizan para trabajar con disyunciones, es decir, proposiciones de la forma $A \vee B$.

En una demostración informal, si queremos demostrar que "A o B" es cierto, nos basta con demostrar una de las dos. Utilizamos `left` para indicar que vamos a demostrar la parte izquierda (A), y `right` si queremos demostrar la parte derecha (B). Por ejemplo:

```
example : (p q : Prop) (hp : p) :
    p ∨ q := by
```

```
Tactic state
  1 goal
  p q : Prop
  hp : p
  ⊢ p ∨ q
```

↓

```
example : (p q : Prop) (hp : p) :
    p ∨ q := by
  left
```

```
Tactic state
  1 goal
  p q : Prop
  hp : p
  ⊢ p
```

Podríamos completar esta demostración aplicando `exact hp`.

`constructor`

Utilizamos `constructor` para trabajar con conjunciones, es decir, proposiciones de la forma $A \wedge B$.

Cuando queremos demostrar que "A y B" es cierto, podemos demostrar A por un lado y B por otro. Al aplicar `constructor`, Lean divide un objetivo `A ∧ B` en dos sub-objetivos con el mismo contexto: uno para `A` y otro para `B`. Por ejemplo:

```
example (p q : Prop) (hp : p)
    (hq : q) : p ∧ q := by
```

```
Tactic state
  1 goal
  p q : Prop
  hp : p
  hq : q
  ⊢ p ∧ q
```

↓

```
example (p q : Prop) (hp : p)
    (hq : q) : p ∧ q := by
  constructor
```

```
Tactic state
  2 goals
  case left
    p q : Prop
    hp : p
    hq : q
    ⊢ p
  case right
    p q : Prop
    hp : p
    hq : q
    ⊢ q
```

Después de aplicar `constructor`, el InfoView mostrará dos objetivos pendientes (`2 goals`). Al resolver cada uno por separado, completamos la demostración.

```
example (p q : Prop) (hp : p)
    (hq : q) : p ∧ q := by
  constructor
  exact hp
  exact hq
```

```
Tactic state
  No goals
```

Aunque lo anterior es correcto, lo habitual cuando trabajamos con más de una tesis es utilizar `·` para separarlas. Cuando escribimos `·` tras un salto de línea, Lean enfoca el primer objetivo, ocultando temporalmente el resto. Por ejemplo:

```
example (p q : Prop) (hp : p)
    (hq : q) : p ∧ q := by
  constructor
  ·
```

```
Tactic state
  1 goal
  case left
    p q : Prop
    hp : p
    hq : q
    ⊢ p
```

Si colocamos el cursor al final, el InfoView solo muestra `1 goal`, porque el segundo objetivo está oculto por ahora. La demostración completa en este estilo sería:

```
example (p q : Prop) (hp : p)
    (hq : q) : p ∧ q := by
  constructor
  · exact hp
  · exact hq
```

```
Tactic state
  No goals
```

`cases'`

La táctica `cases'` se utiliza para analizar una disyunción en el contexto, es decir, una hipótesis de la forma $A \vee B$.

En una demostración informal, equivale a hacer un razonamiento por casos: "Supongamos que ocurre A, veamos si se sigue la tesis; supongamos depués que ocurre B, y comprobemos si también se sigue".

Al aplicar `cases' h` sobre una hipótesis h, Lean duplica el objetivo (que no cambia), pero modifica el contexto en cada uno de los nuevos objetivos, introduciendo las hipótesis correspondientes a cada caso. Utilizamos el comando `with` para asignar nombres a las nuevas hipótesis. Por ejemplo:

```
example (p q : Prop) (h : p ∨ q)
    (hpq : p → q) : q := by
```

```
Tactic state
  1 goal
  p q : Prop
  h : p ∨ q
  hpq : p → q
  ⊢ q
```

↓

```
example (p q : Prop) (h : p ∨ q)
    (hpq : p → q) : q := by
  cases' h with hp hq
```

```
Tactic state
  2 goals
  case inl
    p q : Prop
    hpq : p → q
    hp : p
    ⊢ q
  case inr
    p q : Prop
    hpq : p → q
    hq : q
    ⊢ q
```

Podemos entonces completar la demostración con las herramientas que tenemos hasta ahora:

```
example (p q : Prop) (h : p ∨ q)
    (hpq : p → q) : q := by
  cases' h with hp hq
  · apply hpq
    exact hp
  · exact hq
```

```
Tactic state
  No goals
```

Como hemos visto por medio de estos ejemplos, completar una demostración

en modo táctico consiste en combinar estas instrucciones una después de otra, haciendo que las hipótesis y las tesis vayan avanzando hasta alcanzar el estado deseado: `No goals`. Las tácticas nos dan la flexibilidad necesaria para formalizar una gran variedad de resultados matemáticos.

2.4.5 Herramientas de automatización y búsqueda en Mathlib

A medida que las demostraciones en Lean se vuelven más complejas, no siempre resulta práctico construir cada paso manualmente. Para agilizar el proceso, Lean incorpora algunas herramientas de automatización que permiten delegar ciertas tareas al sistema.

Además, en lugar de volver a demostrar resultados que ya están formalizados, es fundamental **aprovechar la biblioteca matemática de Lean, Mathlib**, que contiene miles de definiciones y teoremas disponibles para su reutilización.

Sin embargo, apoyarse en Mathlib no siempre es directo: los resultados pueden tener nombres poco intuitivos o muy específicos, y encontrar el lema que necesitamos en un momento dado no es siempre fácil.

Por ejemplo, un resultado tan simple como: "Si a, b, c son números reales tales que $a < b$ y $c < 0$, entonces $a + c < b$" (que en una prueba informal consideraríamos casi trivial), aparece en Mathlib con el nombre `add_lt_of_lt_of_neg'`. En la práctica, recordar todos estos nombres resulta inviable, incluso para resultados elementales.

En esta sección introduciremos las tácticas `simp` y `exact?`, que nos ayudan a resolver objetivos simples, y dos herramientas externas que podemos utilizar para localizar resultados en Mathlib. Además, veremos la forma en la que integrar estas herramientas en nuestro proceso de demostración de resultados.

`simp`

La manera más sencilla de apoyarse en la librería de Mathlib es utilizar la táctica `simp`. Esta táctica hace una búsqueda exhaustiva entre una base de datos de lemas de Mathlib que están marcados con el atributo `simp`, intentando simplificar lo máximo posible el objetivo o las hipótesis a las que se aplique.

La táctica `simp` se puede utilizar en cualquier momento de la demostración, pero resulta especialmente útil cuando algo que queremos demostrar parece evidente o suficientemente simple. Por ejemplo:

```
example (G : Type) [Group G] (a b c : G) :
    a * a⁻¹ * 1 * b = b * c * c⁻¹ := by
 simp
```

Solamente usando `simp` podemos terminar la demostración en este caso. Realmente, lo único que hace es reescribir reiteradamente resultados de la forma `A = B` ó `A ↔ B`, hasta que no puede reescribir nada más, de manera mecánica. Por tanto, aunque es útil en muchos casos, en otros es posible que no nos ayude.

En la práctica, cuando nos resulte sencillo utilizar otras tácticas o resultados conocidos, eso será preferible a utilizar `simp`, primero porque al tratarse de una búsqueda exhaustiva, no es una táctica computacionalmente eficiente, y segundo porque empeora la legibilidad del código, ya que a veces es difícil saber cómo ocurren ciertas simplificaciones.

`exact?`

Lean incorpora algunas tácticas que intentan cerrar el objetivo actual utilizando tanto las hipótesis del contexto como los resultados disponibles en los archivos importados. Las más destacadas son `exact?`[13] y `apply?`.

A lo largo del proyecto, la que he utilizado con mayor frecuencia es `exact?`. Esta táctica intenta encontrar una expresión que tenga exactamente el tipo del objetivo actual, buscando tanto en la información local (hipótesis del contexto, resultados definidos anteriormente) como en la librería de Mathlib.

Por ejemplo, en el caso de encontrar hipótesis locales:

```
example (p : Prop) : p → p := by
  intro hp
  exact?
```

```
Suggestions
  Try this: exact hp
```

Y en el caso de encontrar resultados de Mathlib:

```
example (n : ℕ) : n ≥ 0 := by
   exact?
```

```
Suggestions
  Try this: exact
  Nat.zero_le n
```

En general, utilizar la expresión sugerida por `exact?` concluirá la prueba.

A pesar de que `exact?` nos puede ayudar en muchos casos, es una herramienta relativamente sencilla, que solo puede dar un paso (aplicar un teorema o una hipótesis). Esto implica que si no tenemos las hipótesis exactas de los teoremas como aparecen en Mathlib, `exact?` no encontrará ninguna solución.

Cuando trabajamos con hipótesis más complejas, lo habitual no es utilizar

[13] La táctica `exact?` tenía el nombre `library_search` en Lean 3.

`exact?` directamente para probar nuestra tesis, sino para probar ciertos resultados intermedios. Por esto, una táctica crucial a la hora de trabajar con `exact?` es `have`, el equivalente en demostraciones informales a declarar un lema en mitad de una demostración. Por ejemplo, supongamos que queremos probar:

```
example (p q r : Prop) (hpq : p → q) (hqr : q → r) (hp : p) : r
```

En lugar de tratar de demostrar inmediatamente `r`, podríamos probar, de manera intermedia, que se tiene `q`. Para esto utilizamos `have`:

```
example (p q r : Prop) (hpq : p → q)
    (hqr : q → r) (hp : p) : r := by
  have hq : q
```

```
Tactic state
  2 goals
  case hq
    (...)
    ⊢ q
  (...)
  ⊢ r
```

Escribir `have hq : q` introduce una nueva tesis, `q`, independiente de la anterior. Una vez completemos la prueba de esta nueva tesis, podremos usar el resultado en nuestra demostración. Por tanto, podríamos completar el ejemplo anterior de la siguiente forma:

```
example (p q r : Prop) (hpq : p → q) (hqr : q → r) (hp : p) : r := by
  have hq : q
  · apply hpq
    exact hp
  apply hqr
  exact hq
```

Recordemos que utilizamos el punto `·` para separar la demostración de `hq` del resto de la demostración.

Veamos por tanto como es el proceso de trabajar con `exact?`. Consideremos el siguiente ejemplo, para el que `exact?` no encuentra ningún resultado:

```
example (x : ℝ) (hx : x > 0) :
    x / x = 1 := by
  exact?
```

```
Tactic state
  1 goal
  x : ℝ
  hx : x > 0
  ⊢ x / x = 1
Messages
  ‘exact?‘ could not close
  the goal.
```

1. Mirando el estado actual de la demostración, identificar cuál sería una hipótesis que desearíamos tener en nuestro contexto. En este caso, al tratarse de una división, podría ser necesario tener la hipótesis $x \neq 0$.

2. Añadir la nueva tesis utilizando have[14].

```
example (x : ℝ) (hx : x > 0) :
    x / x = 1 := by
  have h : x ≠ 0
  ·
```

```
Tactic state
  1 goal
  x : ℝ
  hx : x > 0
  ⊢ x ≠ 0
```

3. Intentar demostrar la nueva tesis utilizando `exact?`.

```
example (x : ℝ) (hx : x > 0) :
    x / x = 1 := by
  have h : x ≠ 0
  · exact?
```

```
Suggestions
  Try this: Ne.symm
  (ne_of_lt hx)
```

Con esta nueva hipótesis, parece probable que `exact?` sea capaz de terminar la demostración. En efecto:

```
example (x : ℝ) (hx : x > 0) :
    x / x = 1 := by
  have h : x ≠ 0
  · exact Ne.symm (ne_of_lt hx)
  exact?
```

```
Suggestions
  Try this:
  (div_eq_one_iff_eq
  h).mpr rfl
```

La táctica `exact?` es un ejemplo de motor de búsqueda formal: una herramienta que, mediante meta-programación en Lean, compara el objetivo actual con los tipos de todos los lemas disponibles y devuelve aquellos con coincidencias exactas. Por tanto, la clave de usar `exact?` de manera eficaz reside en **desarrollar gradualmente una cierta intuición** sobre qué resultados es probable que estén formalizados en Mathlib, y la forma concreta en la que están formulados.

En efecto, reconocer que un lema de Mathlib sobre división por x probablemente requería la hipótesis $x \neq 0$ (y no simplemente $x > 0$) ha sido esencial para poder aplicar `exact?` con éxito en el ejemplo anterior.

A parte de `exact?`, existen tácticas similares como `apply?` y `rw?`, que funcionan del mismo modo y permiten dar pasos intermedios. Sin embargo, en la práctica estas tácticas suelen devolver una larga lista de opciones, muchas de las cuales no son relevantes o útiles. Por tanto, cuando `exact?` no es suficiente, es más eficaz recurrir a otras herramientas de búsqueda.

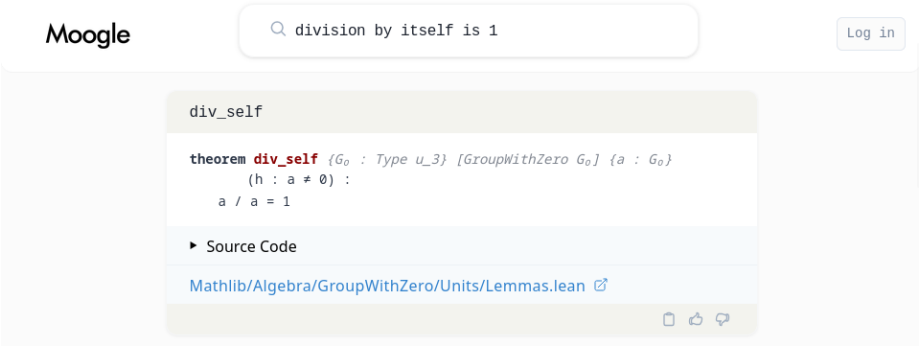
[14] En algunos casos, será más útil escribir aquello que creemos poder necesitar fuera de la demostración, utilizando `example`, porque podremos escribir resultados más generales.

Otras herramientas

A lo largo de este proyecto he utilizado fundamentalmente dos herramientas externas de búsqueda en Mathlib: Moogle [14] y LeanSearch [15]. Ambos son motores de búsqueda semántica, lo que significa que no se limitan a buscar coincidencias literales en el texto, sino que intentan interpretar el significado matemático de nuestra consulta y compararlo con los resultados de Mathlib. Para ello utilizan modelos de lenguaje de gran escala (LLMs), que permiten establecer relaciones entre enunciados aunque estén formulados de distinta manera. En particular, admiten consultas con los siguientes formatos [15]:

- Descripciones en lenguaje natural
- Nombres de teoremas conocidos
- Notación matemática (en LaTeX)
- Código Lean

Por ejemplo, si en el caso anterior no se nos hubiera ocurrido la idea de demostrar primero $x \neq 0$, podríamos haber buscado en Moogle algo del estilo de *"division by itself is 1"*. De hecho, el segundo resultado de esta búsqueda en Moogle es:



Que no es el mismo resultado que proponía `exact?`, pero parece incluso más simple. Podríamos volver a nuestro ejemplo y escribir

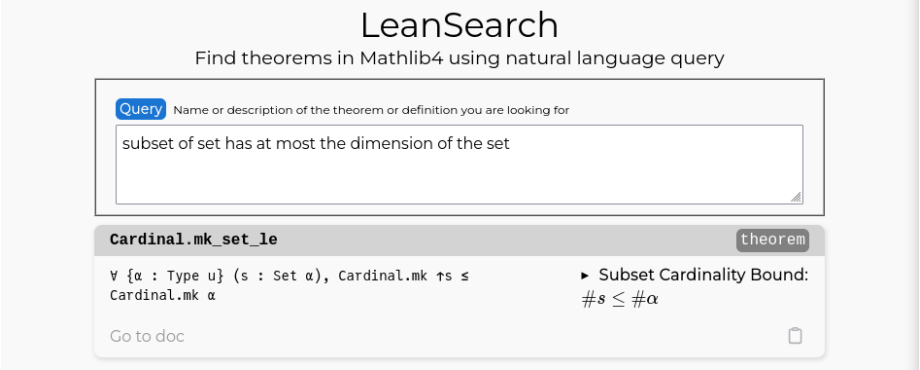
```
example (x : ℝ) (hx : x > 0) :
    x / x = 1 := by
  apply div_self
```

```
Tactic state
  1 goal
  x : ℝ
  hx : x > 0
  ⊢ x ≠ 0
```

Con lo que ya sólo faltaría demostrar que $x \neq 0$.

En general, he encontrado que LeanSearch funciona mejor que Moogle, especialmente en términos de relevancia de los resultados obtenidos. Sin embargo, al principio del proyecto solo conocía Moogle, y descubrí LeanSearch más tarde, por lo que he utilizado Moogle mayoritariamente.

Un ejemplo de una búsqueda real que necesité para el trabajo en LeanSearch, fue *"subset of set has at most the dimension of the set"*.



El primer resultado que aparece es justo el que necesitaba. Sin embargo, antes de buscarlo no tenía ninguna idea de cómo formalizar los resultados en los que estaba trabajando, especialmente porque no estaba familiarizada con el módulo `Cardinal`. Este fue un ejemplo claro de que estas herramientas permiten acceder a partes de Mathlib que de otro modo serían difíciles de localizar.

En conjunto, herramientas como `exact?`, LeanSearch o Moogle han resultado fundamentales para hacer más eficiente el proceso de formalización, permitiendo apoyarse en Mathlib de forma efectiva sin necesidad de conocerla en profundidad desde el principio.

2.4.6 Noncomputable y el axioma de elección

Para finalizar esta sección sobre Lean en la práctica, es útil comentar brevemente una cuestión que aparecerá en algunas de las definiciones posteriores: el uso del **axioma de elección** y la palabra clave `noncomputable`.

En Lean, el axioma de elección se introduce de la siguiente forma:

```
axiom choice {α : Sort u} : Nonempty α → α
```

Es decir, dado un tipo no vacío, `choice` devuelve un elemento de ese tipo, aunque no nos dice cómo encontrarlo. Por este motivo, su uso impide extraer información computable del resultado.

En consecuencia, cuando definimos funciones o construcciones que dependen de

`choice`, Lean nos obliga a marcarlas como `noncomputable`. Un ejemplo es la función `choose`, que dada una prueba de tipo existencial, selecciona un testigo:

```
noncomputable def choose {α : Sort u} {p : α → Prop}
    (h : ∃ x, p x) : α :=
  (indefiniteDescription p h).val
```

A menudo utilizaremos `choose` (`Classical.choose`, ya que se encuentra en el módulo `Classical`) en nuestros resultados, junto con el siguiente lema

```
theorem choose_spec {α : Sort u} {p : α → Prop}
    (h : ∃ x, p x) : p (choose h) :=
  (indefiniteDescription p h).property
```

que es una demostración de que el elemento elegido mediante `choose` cumple las propiedades que le pedíamos.

El uso de `noncomputable` no representa un problema para nosotros (ni, en general, para la comunidad matemática), ya que en este trabajo no nos interesa que las construcciones sean computables: trabajamos con ellas desde un punto de vista lógico y matemático, no algorítmico.

Además, usar el axioma de elección tiene una ventaja práctica: cuando utilicemos `choose`, el elemento elegido será siempre el mismo (aunque no sepamos cuál es), y tendrá siempre la propiedad `choose_spec`. Esto permite trabajar con él de forma coherente dentro de una demostración y referirse a él varias veces como si fuera un objeto determinado.

En contraposición, otra forma de obtener un testigo de una prueba de existencia es utilizar la táctica `obtain`, que se utiliza de la siguiente manera:

```
example : (∃ n : ℕ, n > 3) → ∃ m : ℕ, m > 2 := by
intro h   -- h : ∃ n : ℕ, n > 3
obtain ⟨n, hn⟩ := h    -- n : ℕ, hn : n > 3
...
```

La diferencia fundamental entre utilizar el axioma de elección y utilizar `obtain` es que dos testigos obtenidos mediante obtain del mismo tipo (por ejemplo de tipo `∃ n : nat, n >3`) no serán necesariamente iguales, mientras que si fueron obtenidos mediante `Classical.choose` siempre serán iguales.

3 Espacios topológicos en Lean

En esta sección veremos cómo se representan en Lean algunos conceptos básicos de topología general. El objetivo no es desarrollar la teoría completa, sino mostrar ejemplos concretos de definiciones y demostraciones formales, que sirvan como primer contacto con el trabajo en Lean sobre espacios topológicos.

Las definiciones y resultados matemáticos utilizados son los habituales en topología general. Aunque inicialmente me basé en los apuntes que tomé en la asignatura Topología Elemental, posteriormente los he contrastado con [16] como referencia estándar.

3.1 Espacios topológicos

Definición 3.1 (**Espacio topológico**). *Sea X un conjunto y $\mathcal{T}$ una colección de subconjuntos de X de forma que*

1. *Los conjuntos $\emptyset$ y X pertenecen a $\mathcal{T}$.*
2. *Cualquier intersección finita de elementos de $\mathcal{T}$ pertenece a $\mathcal{T}$.*
3. *Cualquier unión arbitraria de elementos de $\mathcal{T}$ pertenece a $\mathcal{T}$.*

Entonces diremos que $\mathcal{T}$ es una topología *sobre X, que $(X, \mathcal{T})$ es un* espacio topológico *y que los elementos de $\mathcal{T}$ son* abiertos *en este espacio.*

En Lean, esta definición se escribe como una estructura que consta de cuatro elementos:

```
class TopologicalSpace (X : Type u) where
  IsOpen : Set X → Prop
  isOpen_univ : IsOpen Set.univ
  isOpen_inter : ∀ s t, IsOpen s → IsOpen t → IsOpen (s ∩ t)
  isOpen_sUnion : ∀ s, (∀ t ∈ s, IsOpen t) → IsOpen (⋃₀ s)
```

El primer elemento, `IsOpen`, es una función que lleva cada conjunto de X en una proposición, es decir, es una descripción de los elementos de $\mathcal{T}$ como el conjunto `{U ∈ Set (X) | IsOpen U}`. Los otros tres elementos son demostraciones de las propiedades de la definición.

Veamos algunos ejemplos y sus demostraciones en Lean.

Ejemplo 3.1. *Sea X un conjunto cualquiera. Consideremos la colección de todos los subconjuntos de X, $\mathcal{T} = \mathcal{P}(X)$. Entonces $\mathcal{T}$ es una topología sobre X, a la que llamamos* topología discreta.

Demostración. Podemos describir $\mathcal{T}$ como `{U ∈ Set (X) | true}`, porque `IsOpen` es cierto para cualquier U.

```
def DiscreteTopo (X : Type) : TopologicalSpace X where
  IsOpen (_ : Set X) := true
```

Ahora, demostrar el resto de propiedades es sencillo:

```
  isOpen_univ := by
```

```
⊢ (fun x ↦ true = true)
  Set.univ
```

Aplicar la función `fun x ↦ true` a cualquier conjunto retorna siempre `true`, por tanto basta con usar `trivial`.

```
def DiscreteTopo (X : Type) : TopologicalSpace X where
  IsOpen (_ : Set X) := true
  isOpen_univ := by
    trivial
  isOpen_inter := by
    intros
    trivial
  isOpen_sUnion := by
    intros
    trivial
```

□

Ejemplo 3.2. *Sea X un conjunto cualquiera. Consideremos la colección $\mathcal{T} = \{\emptyset, X\}$. Entonces $\mathcal{T}$ es una topología sobre X, a la que llamamos* topología trivial.

Demostración. Podemos describir $\mathcal{T}$ como `{U ∈ Set (X) | U = X ∨ U = ∅}`.

```
def TrivialTopology (X : Type) : TopologicalSpace X where
  IsOpen (s : Set X) := s = Set.univ ∨ s = ∅
```

La primera condición se cumple trivialmente: queremos ver $X = X \vee X = \emptyset$.

```
  isOpen_univ := by
    left -- elegimos X = X
    rfl
```

Consideremos ahora dos abiertos (`intro`). Diferenciamos en casos:

```
  isOpen_inter := by
    intro s t hs ht
    cases' hs with hs_univ hs_empty
```

```
    cases' ht with ht_univ ht_empty
```

Si ambos son X, la intersección será X y por tanto abierta. Si uno de los dos es vacío, entonces la intersección es vacía, también abierta.

```
    · left -- s, t = Set.univ, s ∩ t = Set.univ ?
      rw [hs_univ, ht_univ]
      simp
    · right -- t = ∅, s ∩ t = ∅ ?
      rw [ht_empty]
      simp
    · right -- s = ∅, s ∩ t = ∅ ?
      rw [hs_empty]
      simp
```

Consideremos finalmente una colección arbitraria S de abiertos. Para ver si la unión es abierta, consideramos dos casos distintos: o bien X está en S, en cuyo caso la unión es X, o bien no lo está, en cuyo caso todos los conjuntos de S son el vacío y la unión también lo es.

```
  isOpen_sUnion := by
    intro S hS
    cases' Classical.em (Set.univ ∈ S) with h1 h2

    · left -- h1 : Set.univ ∈ S, ⋃₀S = Set.univ ?
      ext s
      constructor
      · intro hs -- hs : s ∈ S, s ∈ Set.univ ?
        trivial
      · intro hs -- hs : s ∈ Set.univ, s ∈ S ?
        use Set.univ -- utiliza h1 implicitamente

    · right -- h2 : Set.univ ∉ S, ⋃₀S = ∅ ?
      simp -- ∀ s ∈ S, s = ∅ ?
      intro s hs
      specialize hS s hs -- aplicar definicion de abierto
      cases' hS with hS hS -- dividir en casos
      · by_contra -- si s = Set.univ, contradiccion
        rw [hS] at hs
        exact h2 hs -- porque s ∈ S, pero Set.univ ∉ S
      · exact hS -- si s = ∅
```

□

Ejemplo 3.3. *Consideremos la recta real y la definición usual de conjunto abierto en* $\mathbb{R}$*, es decir,* $A \subseteq \mathbb{R}$ *es abierto si y solo si para cada punto* $x \in A$ *existe una bola abierta centrada en* x *enteramente contenida en* A*. Sea* $\mathcal{T}$ *la colección de estos abiertos. Entonces* $\mathcal{T}$ *es una topología sobre* $\mathbb{R}$*, a la que llamamos* topología usual.

Demostración. En Lean, podemos describir este espacio topológico dando sus elementos de la siguiente forma, donde cada objeto esta definido anteriormente.

```
def UsualTopology : TopologicalSpace ℝ where
  IsOpen := Real.IsOpen
  isOpen_univ := Real.isOpen_univ
  isOpen_inter := Real.isOpen_inter
  isOpen_sUnion := Real.isOpen_sUnion
```

La definición de abierto se puede escribir así:

```
def Real.IsOpen (s : Set ℝ) : Prop :=
  ∀ x ∈ s, ∃ δ > 0, ∀ y : ℝ, x - δ < y ∧ y < x + δ → y ∈ s
```

Damos la demostración para la intersección finita. El resto utilizan mecanismos parecidos.

Sean por tanto dos subconjuntos s y t de $\mathcal{T}$. Sea $x \in t \cap s$ y queremos ver que existe una bola abierta centrada en x y contenida en $t \cap s$.

```
lemma Real.isOpen_inter (s t : Set ℝ)
    (hs : IsOpen s) (ht : IsOpen t) : IsOpen (s ∩ t) := by
  intro x hx -- x ∈ s ∩ t, ∃ δ > 0, ∀ (y : ℝ), ... → y ∈ s ∩ t ?
```

Puesto que $x \in s$, existe un $\delta_1 > 0$ (`hδ1`) de forma que $B_{\delta_1}(x) \subseteq s$ (`hs`). Análogamente, existe un $\delta_2 > 0$ (`hδ2`) de forma que $B_{\delta_2}(x) \subseteq t$ (`ht`). Basta con tomar $\delta = \text{mín}\{\delta_1, \delta_2\}$.

```
  obtain ⟨δ1, hδ1, hs⟩ := hs x hx.left
  obtain ⟨δ2, hδ2, ht⟩ := ht x hx.right
  use min δ1 δ2
```

Trivialmente $\delta > 0$.

```
  constructor
  · exact lt_min hδ1 hδ2
```

Para ver que $B_\delta(x) \subseteq s \cap t$, consideramos $y \in B_\delta(x)$ y queremos ver que $y \in s$ y que $y \in t$. Para ver $y \in s$, como $B_{\delta_1}(x) \subseteq s$ (`hs`), basta ver $y \in B_{\delta_1}(x)$.

```
  · intro y hy
    constructor
    · apply hs -- x - δ1 < y ∧ y < x + δ1 ?
```

En realidad, esta condición se reduce a dos inecuaciones, que son fáciles de probar contando con que $\delta \leq \delta_1$ (`hδ`).

```
    have hδ := min_le_left δ1 δ2
    constructor
    all_goals linarith
```

Probar que $y \in t$ es análogo.

□

3.1.1 Conjuntos abiertos

Como hemos dicho, los conjuntos abiertos en un espacio topológico son los elementos de la topología. En Lean, es una función `TopologicalSpace.IsOpen` de tipo `Set X → Prop`. Podemos utilizar esta definición directamente para demostrar que un abierto lo es.

Ejemplo 3.4. *Por definición, el universo, X, siempre es abierto. En efecto:*

```
example (X : Type) [T : TopologicalSpace X] : T.IsOpen Set.univ := by
  exact TopologicalSpace.isOpen_univ
```

Ejemplo 3.5. *En la Definición 3.1, teníamos la condición $\emptyset \in \mathcal{T}$, condición que no aparece en la definición de Mathlib.*

Se puede probar que el vacío es abierto a partir del resto de condiciones: podemos escribir $\emptyset$ como $\emptyset = \bigcup_{x\in\emptyset}\{x\}$. Aplicando que la unión arbitraria de abiertos es abierta, bastaría ver que $\forall x \in \emptyset$, $\{x\}$ es un conjunto abierto. Lo cual es trivial.

```
example {X : Type} [T : TopologicalSpace X] :
    IsOpen (∅ : Set X) := by
  rw [← Set.sUnion_empty] -- ∅ = ⋃₀ ∅
  apply isOpen_sUnion
  intro t ht -- ht : t ∈ ∅, IsOpen t ?
  exfalso
  exact ht -- t ∈ ∅ = False
```

Ejemplo 3.6. *En la topología usual, $(\mathbb{R}, \mathcal{T}_u)$, los intervalos abiertos $I = (a, b)$ con $a < b$ son abiertos de la topología.*

Demostración. Consideremos un intervalo de la forma (a, b) y queremos ver que es abierto. Para ello, sea $x \in (a, b)$, y veamos que existe un $\delta > 0$ tal que $\forall y \in \mathbb{R}$, si $y \in B_\delta(x)$ entonces $y \in (a, b)$.

```
lemma ioo_open_in_R (a b : ℝ) :
    UsualTopology.IsOpen ((Set.Ioo a b) : Set ℝ) := by
  rw [UsualTopology]
  intro x hx
```

Tomamos $\delta = min\{x-a, b-x\}$. Obviamente $\delta > 0$ pues $a < x < b$.

```
  use min (x-a) (b-x)  -- nuestro δ
  constructor
  · simp
    exact hx
```

Sea ahora $y \in B_\delta(x)$ y queremos ver que $y \in (a,b)$. Hay dos posibles casos, según el valor que tome δ. Si $\delta = x - a$, es decir, $x - a < b - x$, entonces se tiene

$$y \in B_\delta(x) \implies x - (x-a) < y < x + (x-a) < x + (b-x) \implies a < y < b,$$

luego $y \in (a,b)$. El caso $\delta = b - x$ es análogo.

```
  · intro y hy
    have cases := lt_or_le (x - a) (b - x)
    cases' cases with h h -- dividimos en casos
    all_goals -- aunque en ambos casos se procede igual
      try rw [min_eq_left_of_lt h] at hy
      try rw [min_eq_right h] at hy
      simp at hy
      constructor
      all_goals linarith
```

□

Definición 3.2 (**Entorno abierto**). *Sea $(X, \mathcal{T})$ un espacio topológico y $x \in X$. Un* entorno abierto *de x en X es un conjunto abierto $U \in \mathcal{T}$ de forma que $x \in U$.*

```
def OpenNeighbourhood {X : Type} [TopologicalSpace X]
  (U : Set X) (x : X) : Prop :=
  x ∈ U ∧ IsOpen U
```

Ejemplo 3.7. *El universo, X, es entorno abierto de cualquier punto $x \in X$. En efecto:*

```
example {X : Type} [TopologicalSpace X] (x : X) :
    OpenNeighbourhood Set.univ x := by
  constructor
  · trivial -- x ∈ Set.univ ?
  · exact isOpen_univ -- IsOpen Set.univ ?
```

Definición 3.3 (**Entorno**). *Sea $(X, \mathcal{T})$ un espacio topológico y $x \in X$. Un* entorno *de x en X es un conjunto $V \subseteq X$ de forma que existe un entorno abierto de x, $U \in \mathcal{T}$, con $U \subseteq V$.*

```
def Neighbourhood {X : Type} [TopologicalSpace X]
  (V : Set X) (x : X) : Prop :=
  ∃ U : Set X, U ⊆ V ∧ OpenNeighbourhood U x
```

Ejemplo 3.8. *Un entorno abierto es también un entorno. En efecto:*

```
example {X : Type} [TopologicalSpace X] (U : Set X) (x : X) :
    OpenNeighbourhood U x → Neighbourhood U x := by
  intro hU
  use U -- usamos el propio U como abierto,
      -- lo que completa la prueba directamente
```

Proposición 3.1 (**Caracterización de conjuntos abiertos**). *Sea $(X, \mathcal{T})$ un espacio topológico y $A \subseteq U$ un conjunto cualquiera. A es abierto si y solo si es entorno de todos sus puntos.*

```
lemma A_open_iff_neighbourhood_of_all {X : Type}
    [T : TopologicalSpace X] {A : Set X} :
    IsOpen A ↔ ∀ x ∈ A, Neighbourhood A x := by
```

Demostración. Demostramos cada implicación separadamente.

```
  constructor; all_goals intro h
```

($\Longrightarrow$) La primera implicación es sencilla: si A es abierto, para cada $x \in A$ basta tomar A como entorno de x.

```
  · intro x hx -- hx : x ∈ A
    use A
    constructor
    · trivial -- A ⊆ A
    · constructor
      · exact hx -- x ∈ A
      · exact h -- IsOpen A
```

($\Longleftarrow$) El recíproco es más complicado. Sabemos que para cada $a \in A$ existe U_a entorno de a. Primero probaremos que

$$A = \bigcup_{a \in A} U_a$$

```
  · have hUnion : A = ⋃ x : A, Classical.choose (h x x.property)
```

Para ello, probamos ambas inclusiones. Si $x \in A$, entonces por nuestra hipótesis existe un entorno de x, U_x. Y por la definición de entorno, eso quiere decir que $x \in U_x$. Luego $x \in \bigcup_a U_a$.

```
    · ext x; constructor; all_goals intro hx
      · have ⟨_, hUx⟩  := Classical.choose_spec (h x hx)
          -- hUx : OpenNeigh. Ux x
        simp -- ∃ a ∈ A, x ∈ Ua ?
        use x, hx
        exact hUx.left -- x ∈ Ux
```

Ahora, si $x \in \bigcup_a U_a$, entonces existe un $a \in A$ con $x \in U_a$ y U_a entorno abierto de a con $U_a \subseteq A$. Luego $x \in U_a \subseteq A$.

```
      · simp at hx
        obtain ⟨a, ha, hx⟩ := hx -- hx : x ∈ Ua
        have ⟨ha', _⟩ := Classical.choose_spec (h a ha)
          -- ha' : Ua ⊆ A
        apply ha'
        exact hx
```

Entonces hemos probado que A se expresa como una unión de conjuntos U_a. Pero sabemos que todos estos conjuntos son entornos abiertos, luego son abiertos. Basta aplicar que la unión de abiertos es abierta.

```
    rw [hUnion]
    apply isOpen_iUnion
    intro a
    exact (Classical.choose_spec (h a a.property)).right.right
```

□

Definición 3.4 (**Interior**). *Sea $(X, \mathcal{T})$ un espacio topológico y $A \subseteq X$. Definimos el* interior *como el conjunto*

$$\overset{\circ}{A} = \bigcup \{U \subseteq X | U \text{ es abierto y } U \subseteq A\}$$

```
def interior (s : Set X) : Set X :=
  ⋃₀ { t | IsOpen t ∧ t ⊆ s }
```

Veamos varias propiedades del interior de un conjunto.

Proposición 3.2. *Sea $(X, \mathcal{T})$ un espacio topológico y $A \subseteq X$. Entonces*

$$\overset{\circ}{A} \subseteq A$$

En Mathlib, este resultado recibe el nombre de `interior_subset`.

Demostración. Sea $a \in \overset{\circ}{A}$. Entonces A es entorno de a y existe un abierto con $a \in U \subseteq A$. Luego $a \in A$.

```
example {X : Type} [T : TopologicalSpace X] (A : Set X) :
    interior A ⊆ A := by
  intro a ha
  obtain ⟨U, hU, ha⟩ := ha -- hU : IsOpen U ∧ U ⊆ A, ha : a ∈ U
  apply hU.right
  exact ha
```

□

Proposición 3.3. *Sea $(X, \mathcal{T})$ un espacio topológico y $A \subseteq X$. Entonces $\overset{\circ}{A}$ es un conjunto abierto.*

En Mathlib, este resultado recibe el nombre de `isOpen_interior`.

Demostración. Por la caracterización de conjuntos abiertos (3.1), basta ver que dado $a \in \overset{\circ}{A}$, $\overset{\circ}{A}$ es entorno de a.

Si $a \in \overset{\circ}{A}$, entonces existe abierto U con $a \in U \subseteq A$. Usamos este U para demostrar que $\overset{\circ}{A}$ es entorno de a.

```
example {X : Type} [T : TopologicalSpace X] (A : Set X) :
    IsOpen (interior A) := by
  apply A_open_iff_neighbourhood_of_all.mpr
  intro a ha
  obtain ⟨U, hU, ha⟩ := ha -- hU : IsOpen U ∧ U ⊆ A, ha : a ∈ U
  use U
  constructor
  · intro x hx -- U ⊆ interior A ?
    use U
  · constructor
    · exact ha -- a ∈ U
    · exact hU.left -- IsOpen U
```

□

Proposición 3.4. *Sea $(X, \mathcal{T})$ un espacio topológico y $A \subseteq X$. Entonces A es abierto si y solo si A es igual a su interior.*

En Mathlib, este resultado recibe el nombre de `interior_eq_iff_isOpen`.

Demostración. El recíproco es trivial, pues ya hemos visto que el interior de un conjunto es abierto.

```
example {X : Type} [T : TopologicalSpace X] (A : Set X) :
    IsOpen A ↔ interior A = A:= by
```

```
  constructor; swap; all_goals intro h
  · rw [← h]
    exact isOpen_interior
```

Ahora, supongamos que A es abierto. Ya hemos visto que $\overset{\circ}{A} \subseteq A$, luego basta ver el otro contenido. Sea $a \in A$. Como A es abierto, es un entorno abierto de a con $A \subseteq A$. Luego $a \in \overset{\circ}{A}$.

```
  · apply Set.Subset.antisymm
    · exact interior_subset
    · intro a ha
      use A
      constructor
      · simp
        exact h
      · exact ha
```

□

3.1.2 Conjuntos cerrados

Definición 3.5 (**Conjunto cerrado**). *Sea $(X, \mathcal{T})$ un espacio topológico y $A \subseteq X$. Decimos que A es* cerrado *en X si A^c es abierto en X.*

```
class IsClosed (s : Set X) : Prop where
  isOpen_compl : IsOpen s^c
```

Ejemplo 3.9. *El universo es cerrado, porque el vacío es abierto. El vacío es cerrado, porque el universo es cerrado.*

```
example (X : Type) [TopologicalSpace X] :
    IsClosed (Set.univ : Set X) := by
  rw [← isOpen_compl_iff] -- IsOpen X^c ?
  rw [Set.compl_univ] -- IsOpen ∅ ?
  exact isOpen_empty
```

Ejemplo 3.10. *La intersección arbitraria de cerrados es cerrada. La unión finita de cerrados es cerrada. Ambas se deducen de manera sencilla de la definición de espacio topológico y de conjunto cerrado.*

```
example (X : Type) [TopologicalSpace X] (A B : Set X) (hA : IsClosed A)
    (hB : IsClosed B) : IsClosed (A ∪ B) := by
rw [← isOpen_compl_iff] at *
rw [Set.compl_union] -- usar (A ∪ B)^c = (A^c ∩ B^c)
```

```
  apply TopologicalSpace.isOpen_inter
  exact hA
  exact hB
```

Definición 3.6 (**Clausura**). *Sea $(X, \mathcal{T})$ un espacio topológico y $A \subseteq X$. Definimos la* clausura *de A como el conjunto*

$$\overline{A} = \bigcap \{K \subseteq X | K \text{ es cerrado y } A \subseteq K\}$$

```
def closure (s : Set X) : Set X :=
  ⋂₀ { t | IsClosed t ∧ s ⊆ t }
```

Veamos algunas propiedades de la clausura de un conjunto.

Proposición 3.5. *Sea $(X, \mathcal{T})$ un espacio topológico y $A \subseteq X$. Entonces*

$$A \subseteq \overline{A}$$

En Mathlib, este resultado recibe el nombre de `subset_closure`.

Demostración. Sea $a \in A$ y queremos ver que $a \in \overline{A}$. Como $\overline{A}$ es una intersección, esto es equivalente a probar que para cada K cerrado con $A \subseteq K$, $x \in K$. Pero esto es trivial porque, para cada uno de esos K, $x \in A \subseteq K$.

```
example {X : Type} [T : TopologicalSpace X] (A : Set X) :
    A ⊆ closure A := by
  intro x hx
  intro K hK -- hK : IsClosed K ∧ A ⊆ K; x ∈ K ?
  apply hK.right -- x ∈ A ?
  exact hx
```

□

Proposición 3.6. *Sea $(X, \mathcal{T})$ un espacio topológico y $A \subseteq X$. Entonces*

$$(\overline{A})^c = \overbrace{(A^c)}^{\circ}$$

En Mathlib, este resultado recibe el nombre de `interior_compl`.

Demostración. Veamos ambos contenidos por separado.

```
example {X : Type} [T : TopologicalSpace X] (A : Set X) :
    (closure A)ᶜ = interior (Aᶜ) := by
  ext x; constructor; all_goals intro hx
```

($\subseteq$) Supongamos que $x \in (\overline{A})^c$, es decir $x \notin \overline{A}$. Esto quiere decir que existe un K cerrado de forma que $A \subseteq K$ y $x \notin K$.

```
· simp [closure] at hx
  obtain ⟨K, hKclosed, hKA, hKx⟩ := hx -- hKx : x ∉ K
```

Para ver que x está en el interior de A^c, queremos ver que existe un abierto contenido en A^c que contiene a x. Consideremos el abierto K^c. Como $A \subseteq K$, se tiene $K^c \subseteq A^c$, y como $x \notin K$, se tiene $x \in K^c$

```
  use Kᶜ
  constructor
  · constructor
    · exact isOpen_compl_iff.mpr hKclosed -- IsOpen Kᶜ
    · exact Set.compl_subset_compl_of_subset hKA -- Kᶜ ⊆ Aᶜ
  · exact hKx -- x ∈ Kᶜ
```

($\supseteq$) Sea x en el interior de A^c. Entonces existe un abierto U con $x \in U \subseteq A^c$. Para ver que x está en el complementario de $\overline{A}$, supongamos, por reducción al absurdo, que $x \in \overline{A}$.

```
· obtain ⟨U, hU, hUx⟩ := hx
  obtain ⟨hUopen, hUA⟩ := hU
  by_contra hx -- hx : x ∈ closure A
```

En ese caso, para cada K cerrado con $A \subseteq K$, se tiene $x \in K$. En particular, como U^c es cerrado por ser U abierto y tiene $A \subseteq U^c$ por ser $U \subseteq A^c$, se tiene que $x \in U^c$. Pero $x \in U$, lo cual es una contradicción.

```
  simp [closure] at hx
  specialize hx Uᶜ
    (by exact isClosed_compl_iff.mpr hUopen)
    (by exact Set.subset_compl_comm.mp hUA)
  exact hx hUx
```

□

Proposición 3.7. *Sea $(X, \mathcal{T})$ un espacio topológico y $A \subseteq X$. Entonces $\overline{A}$ es un conjunto cerrado.*

En Mathlib, este resultado recibe el nombre de `isClosed_closure`.

Demostración. La prueba es sencilla utilizando el resultado anterior: $\overline{A}$ es cerrado si $(\overline{A})^c$ es abierto. Pero hemos visto que $(\overline{A})^c = (A^c)^\circ$, y el interior de cualquier conjunto es abierto.

```
example {X : Type} [T : TopologicalSpace X] (A : Set X) :
    IsClosed (closure A) := by
  apply isOpen_compl_iff.mp -- IsOpen (closure A)ᶜ ?
  rw [← interior_compl] -- IsOpen (interior Aᶜ) ?
  exact isOpen_interior
```

□

Proposición 3.8. *Sea $(X, \mathcal{T})$ un espacio topológico y $A \subseteq X$. Entonces A es cerrado si y solo si es igual a su clausura.*

Demostración. El recíproco es trivial, pues ya hemos visto que la clausura de un conjunto es cerrada.

```
example {X : Type} [T : TopologicalSpace X] (A : Set X) :
    IsClosed A ↔ closure A = A := by
  constructor; swap; all_goals intro h
  · rw [← h]
    exact isClosed_closure
```

Ahora, supongamos que A es cerrado. Entonces A^c es abierto, luego es igual a su interior. Pero el interior de A^c hemos visto que es $(\overline{A})^c$, luego $A^c = (\overline{A})^c$, de lo que se deduce $A = \overline{A}$.

```
  · rw [← isOpen_compl_iff, ← interior_eq_iff_isOpen,
  interior_compl] at h
    rw [← compl_compl A, ← h, compl_compl]
    exact closure_closure
```

□

3.2 Bases

Definición 3.7. *Sea $\mathcal{T}$ una topología. Una* base de la topología $\mathcal{T}$ *es una colección de abiertos $\mathcal{B} \subset \mathcal{T}$ de forma que cada abierto de $\mathcal{T}$ es unión de abiertos de $\mathcal{B}$*[15]*.*

```
def isTopoBase {X : Type} [TopologicalSpace X]
    (B : Set (Set X)) : Prop :=
  (∀ U ∈ B, IsOpen U) ∧
  (∀ V : Set X, IsOpen V → ∃ UB ⊆ B, V = ⋃₀ UB)
```

[15]Hay varias formas de dar esta definición, esta es la que yo he elegido para definirla en Lean, de manera independiente a Mathlib. Por tanto, los resultados de esta sección no se encuentran literalmente en Mathlib.

Ejemplo 3.11. *El conjunto de los intervalos abiertos en* $\mathbb{R}$*,*

$$\mathcal{B} = \{I = (a, b) \mid a < b\},$$

es una base de la topología usual $(\mathbb{R}, \mathcal{T}_u)$*.*

Demostración. Hemos visto que los intervalos abiertos son abiertos en la topología usual (Ejemplo 3.6). Por tanto la primera parte de la definición de base ya la tenemos.

```
lemma BaseOfRealTopo [T : TopologicalSpace ℝ] (hT : T =
  UsualTopology) :
    isTopoBase {s | ∃ a b : ℝ, s = Set.Ioo a b} := by
  constructor
  · intro U hU -- sea U en el conjunto
    obtain ⟨a, b, hU⟩ := hU -- hU : U = (a, b)
    rw [hU, hT]
    exact ioo_open_in_R a b -- aplicamos el resultado anterior
```

Ahora, para la segunda parte, consideramos un $U \subseteq X$ cualquiera. Queremos ver que se escribe como unión de intervalos abiertos.

```
  · intro U hUopen
    rw [hT] at hUopen
```

Para ello, consideremos para cada $x \in U$, el δ_u resultante de aplicar la definición de abierto, es decir, tal que $B_{\delta_x}(x) \subseteq U$. Queremos demostrar que $U = \bigcup\{B_{\delta_x}(x) | x \in U\}$, y que este es un subconjunto de $\mathcal{B}$.

```
    let δ : U → ℝ := fun x ↦ Classical.choose (hUopen x x.property)
    have δspec : ∀ x : U, 0 < δ x
        ∧ ∀ y : ℝ, ↑x - δ x < y ∧ y < ↑x + δ x → y ∈ U :=
      fun x ↦ Classical.choose_spec (hUopen x (x.property))

    use {s | ∃ x, s = Set.Ioo (x - δ x) (x + δ x)}
```

Obviamente, es un subconjunto de $\mathcal{B}$ puesto que es un conjunto formado por intervalos abiertos.

```
    constructor
    · intro V hV
      obtain ⟨x, hV⟩ := hV
      use (↑x - δ x), (↑x + δ x)
```

Ahora, para ver que $U = \bigcup\{B_{\delta_x}(x) | x \in U\}$, demostramos las dos inclusiones de manera separada.

```
    · ext u; constructor; all_goals intro hu
```

($\subseteq$) Sea $u \in U$. Entonces $u \in B_{\delta_u}$ trivialmente. Luego está en la unión.

```
· use Set.Ioo (↑u - δ ⟨u, hu⟩) (↑u + δ ⟨u, hu⟩) -- = Bδu(u)
  constructor
  · simp -- Bδu(u) es un intervalo abierto ?
    use u, hu
  · simp -- u ∈ Bδu(u) ?
    exact (δspec ⟨u, hu⟩).left
```

($\supseteq$) Sea $u \in \bigcup\{B_{\delta_x}(x) | x \in U\}$. Entonces existe un $v \in U$ de forma que $u \in B_{\delta_v}(v) \subseteq U$. Luego $u \in U$.

```
· obtain ⟨I, hI, hu⟩ := hu
  obtain ⟨v, hI⟩ := hI
  rw [hI] at hu
  exact (δspec v).right u hu
```

□

3.3 Topología relativa

Definición 3.8 (**Topología relativa**). *Sea $(X, \mathcal{T})$ un espacio topológico y $A \subseteq X$ un subconjunto. Entonces la colección*

$$\mathcal{T}|_A = \{U \cap A | U \in \mathcal{T}\}$$

es una topología sobre A, llamada la topología relativa *a X de A.*

Diremos que $(A, \mathcal{T}|_A)$ es un subespacio topológico *de $(X, \mathcal{T})$.*

En Lean, para poder utilizar esta definición, tenemos que demostrar que este conjunto es, en efecto, una topología.

Demostración. Sea $(X, \mathcal{T})$ un espacio topológico y $A \subseteq X$ un subconjunto, y definimos la colección

$$\mathcal{T}|_A = \{U \cap A | U \in \mathcal{T}\}$$

```
def TopoSubspace {X : Type} (T : TopologicalSpace X) (Y : Set X) :
    TopologicalSpace Y where
  IsOpen (V : Set Y) := ∃ U : Set X, T.IsOpen U ∧ V = U ∩ Y
```

(1) El universo es abierto. En efecto, pues $A = X \cap A$ y X es abierto en X.

```
  isOpen_univ := by
    use (Set.univ : Set X)
    constructor
    · exact T.isOpen_univ
    · simp
```

(2) Sean V_1 y V_2 abiertos en A. Entonces $V_1 = U_1 \cap A$ y $V_2 = U_2 \cap A$ con U_1, U_2 abiertos en X. Luego

$$V_1 \cap V_2 = (U_1 \cap A) \cap (U_2 \cap A) = (U_1 \cap U_2) \cap A,$$

y $U_1 \cap U_2$ es abierto en X por la segunda propiedad.

```
isOpen_inter := by
  intro V1 V2 h1 h2
  obtain ⟨U1, h1open, h1inter⟩ := h1
  obtain ⟨U2, h2open, h2inter⟩ := h2
  use U1 ∩ U2
  constructor
  · exact T.isOpen_inter U1 U2 h1open h2open
  · simp
    rw [h1inter, h2inter]
    exact Eq.symm (Set.inter_inter_distrib_right U1 U2 Y)
```

(3) Sea $S = \{V_i\}_i$ una colección de abiertos en A. Entonces, para cada V_i existe un U_i abierto en X de forma que $V_i = U_i \cap A$. Entonces

$$\bigcup S = \bigcup_i V_i = \bigcup_i (U_i \cap A) = A \cap \bigcup_i U_i,$$

y $\bigcup_i U_i$ es abierto en X por la tercera propiedad. No se incluye aquí la demostración en Lean debido a su mayor complejidad. □

Ejemplo 3.12. *Consideremos $\mathbb{R}$ con la topología usual y el intervalo $[0,1] \subset \mathbb{R}$. En la topología de $[0,1]$ inducida por la topología usual, los intervalos de la forma $[0,b)$ son abiertos para todo $0 < b \leq 1$ (aunque no lo sean en $\mathbb{R}$). También son abiertos los intervalos de la forma $(a,1]$ para cada $0 \leq a < 1$.*

Demostración. Para cada $b > 0$, basta con usar por ejemplo el intervalo abierto $(-1, b)$. Ya hemos visto que los intervalos abiertos son abiertos en $\mathbb{R}$, y se tiene $(-1,b) \cap [0,1] = [0,b)$. Luego $[0,b)$ es abierto en $[0,1]$.

```
lemma ico_open_in_Icc01 {Y : Set ℝ} {hY : Y = Set.Icc 0 1}
    {R : TopologicalSpace Y} {hR : R = TopoSubspace UsualTopology Y}
    (b : ℝ) (hb : 0 < b ∧ b < 1) :
    R.IsOpen ({y | (y : ℝ) ∈ Set.Ico 0 b} : Set Y) := by

rw [hR] -- usar la topologia relativa
rw [UsualTopology] -- usar la def. de topologia usual
use ((Set.Ioo (-1) b) : Set ℝ) -- usar [-1, b)
constructor
· exact ioo_open_in_R (-1) b -- es abierto
· ext x; constructor -- (-1, b) ∩ [0, 1] = [0, b) ?
  all_goals
    intro hx
```

```
    simp at * -- convertirlo todo a inecuaciones
    constructor
    · simp [hY] at hx
      constructor
      all_goals linarith
    · exact hx.right
```

□

3.4 Continuidad

Definición 3.9. *Sean X e Y dos espacios topológicos y $f : X \to Y$ una función entre ambos. Entonces f es* continua *en un punto $x_0 \in X$ si para cada entorno V de $f(x_0)$ en Y, se tiene que $f^{-1}(V)$ es entorno de x_0 en X. Diremos que f es* continua *en X si es continua en cada punto.*

Proposición 3.9 (Caracterización de funciones continuas). *Sean X e Y dos espacios topológicos y $f : X \to Y$ una función entre ambos. Entonces f es continua si y solo si para cada $V \subseteq Y$ abierto, se tiene que $f^{-1}(V)$ es abierto en X.*

Demostración. Veamos ambas implicaciones por separado.

```
example (X Y : Type) [TopologicalSpace X] [TopologicalSpace Y]
    (f : X → Y) :
    (∀ x : X, ∀ V : Set Y, Neighbourhood V (f x) →
      Neighbourhood (f ⁻¹' V) x)
    ↔ ∀ (V : Set Y), IsOpen V → IsOpen (f ⁻¹' V) := by
  constructor; all_goals intro h
```

($\Longrightarrow$) Sea $f : X \to Y$ continua y sea $V \subseteq Y$ un conjunto abierto. Queremos ver que $f^{-1}(V)$ es abierto. Para ello, basta ver que es entorno de todos sus puntos (Prop. 3.1).

Sea $x \in f^{-1}(V)$. Entonces V es entorno de $f(x)$, por ser V un abierto con $f(x) \in V$. Luego, por la definición de continuidad, $f^{-1}(V)$ es entorno de x.

```
  · intro V hVopen
    apply A_open_iff_neighbourhood_of_all.mpr
    intro x hx
    exact h x V
      (by use V; simp; exact ⟨hx, hVopen⟩) -- V entorno de f x
```

($\Longleftarrow$) Sea ahora $x \in X$ y $V \subseteq Y$ un entorno de $f(x)$ en Y. Queremos ver que $f^{-1}(V)$ es entorno de x en X.

Existe un entorno abierto $U \subseteq V$ de $f(x)$ en Y. Entonces $x \in f^{-1}(U) \subseteq f^{-1}(V)$, y es un abierto, por hipótesis. Por tanto $f^{-1}(V)$ es entorno de x.

```
  · intro x V hV
    obtain ⟨U, hUV, hU⟩ := hV
    obtain ⟨hUx, hUopen⟩ := hU
    use f ⁻¹' U
    constructor
    · intro u hu -- f ⁻¹' U ⊆ f ⁻¹' V ?
      apply hUV
      exact hu
    · constructor
      · exact hUx -- x ∈ f ⁻¹' U
      · exact h U hUopen -- IsOpen (f ⁻¹' U)
```

□

Puesto que ambas definiciones son equivalentes, en Mathlib se utiliza la segunda para definir las funciones continuas, y en general utilizaremos esta definición.

```
structure Continuous (f : X → Y) : Prop where
  isOpen_preimage : ∀ s, IsOpen s → IsOpen (f ⁻¹' s)
```

Utilizaremos `continuous_def` para re escribir `Continuous f` por esta definición cuando lo necesitemos.

Ejemplo 3.13. *Sea $f : (X, \mathcal{T}_{disc}) \to (Y, \mathcal{T})$ una función en la que el espacio de salida tiene la topología discreta (Ej. 3.1). Entonces f es continua.*

Si tomamos cualquier abierto de Y, su preimagen será abierta en la topología discreta trivialmente, puesto que cualquier conjunto lo es.

```
lemma continuous_from_discrete {X Y : Type} [T : TopologicalSpace X]
    [TopologicalSpace Y] (h : T = DiscreteTopo X) (f : X → Y) :
  Continuous f := by
  rw [continuous_def]
  intro U _ -- ni si quiera importa si U es abierto
  rw [h, DiscreteTopo] -- utilizamos la def de topologia discreta
  trivial
```

Ejemplo 3.14. *Sea $f : (X, \mathcal{T}) \to (Y, \mathcal{T}_{trivial})$ una función en la que el espacio de llegada tiene la topología trivial (Ej. 3.2). Entonces f es continua.*

Puesto que las únicas posibilidades de abiertos a tomar en Y son el propio Y y el conjunto vacío, sus preimágenes serán respectivamente X y el conjunto vacío, que son abiertos.

```
lemma continuous_to_trivial {X Y : Type} [TopologicalSpace X]
    [T : TopologicalSpace Y] (h : T = TrivialTopology Y) (f : X → Y) :
    Continuous f := by
  rw [continuous_def]
  intro U hU
  rw [h, TrivialTopology] at hU
  cases' hU with hUuniv hUempty
  · rw [hUuniv] -- si U = Y
    exact isOpen_univ
  · rw [hUempty] -- si U = ∅
    exact isOpen_empty
```

Proposición 3.10. *La composición de funciones continuas entre espacios topológicos es también una función continua.*

En Mathlib, este resultado recibe el nombre de `Continuous.comp`.

Demostración. Sean $f : X \to Y$ y $g : Y \to Z$ dos funciones continuas y consideremos su composición $g \circ f : X \to Z$.

Sea W un abierto de Z. Como g es continua, $V = g^{-1}(W)$ es abierto en Y. Como f es continua, $f^{-1}(V)$ es abierto en X. Pero

$$f^{-1}(V) = f^{-1}(g^{-1}(W)) = (g \circ f)^{-1}(W)$$

```
example (X Y Z : Type) [TopologicalSpace X] [TopologicalSpace Y]
    [TopologicalSpace Z] (f : X → Y) (g : Y → Z)
    (hf : Continuous f) (hg : Continuous g) : Continuous (g ∘ f) := by
  rw [continuous_def] at *
  intro W hW
  specialize hg W hW
  specialize hf (g ⁻¹' W) hg
  exact hf
```

□

Proposición 3.11 (Caracterización de la continuidad por cerrados)**.** *Sean X e Y dos espacios topológicos y $f : X \to Y$ una función entre ambos. Entonces f es continua si y solo si para cada $C \subseteq Y$ cerrado, se tiene que $f^{-1}(C)$ es cerrado en X.*

En Mathlib, este resultado recibe el nombre de `continuous_iff_isClosed`.

Demostración. Supongamos que f es continua. Sea C cerrado en Y. Entonces C^c es abierto en Y. Por ser f continua, $f^-1(C^c)$ es abierto en X. Pero $f^-1(C^c) = f^-1(C)^c$, luego $f^-1(C)$ es cerrado en X. El recíproco es análogo.

```
example (X Y : Type) [TopologicalSpace X] [TopologicalSpace Y]
    (f : X → Y) : Continuous f ↔
      ∀ C : Set Y, IsClosed C → IsClosed (f ⁻¹' C) := by
  constructor; all_goals intro h
  · rw [continuous_def] at h
    intro C hC
    rw [← isOpen_compl_iff] at *
    exact h Cᶜ hC
  · rw [continuous_def]
    intro U hU
    rw [← isClosed_compl_iff] at *
    exact h Uᶜ hU
```

□

Proposición 3.12 (Continuidad del subespacio)**.** *Sean X e Y espacios topológicos y Z un subespacio topológico de Y. Una función $f : X \to Z$ es continua si y solo si lo es como función $f : X \to Y$.*

Es decir, para demostrar la condición de continuidad de $f : X \to Z$, basta con tomar abiertos arbitrarios de Y.

Demostración. Veamos cada implicación por separado.

```
lemma continuousInSubspace_iff_trueForSpace {X Y : Type} {Z : Set Y}
    [TX : TopologicalSpace X] [TY : TopologicalSpace Y]
    [TZ : TopologicalSpace Z] (hZ : TZ = TopoSubspace TY Z)
    (f : X → Z) : Continuous f ↔ ∀ U : Set Y,
      TY.IsOpen U → TX.IsOpen (f ⁻¹' (Subtype.val ⁻¹' U)) := by
  rw [continuous_def]
  constructor
  all_goals intro h U hU
```

($\Longrightarrow$) Supongamos que $f : X \to Z$ es continua. Sea U un abierto de Y y queremos ver que $f^{-}1(U)$ es abierto en X. Puesto que f es continua, basta ver que $f(f^{-1})(U)$ es abierto. Pero $f(f^{-1})(U) = U \cap Y$, que es abierto por la definición de topología relativa por ser U abierto.

```
  · apply h
    rw [hZ]
    use U
    constructor
    · exact hU
    · simp
      exact Set.inter_comm Z U
```

($\Longleftarrow$) Supongamos ahora que $f : X \to Y$ es continua. Sea U un abierto en Z. Entonces, por la definición de la topología relativa, existe un V abierto en Y de forma que $U = V \cap Z$. Por ser f continua, $f^{-1}(V)$ es abierto. Entonces $f^{-1}(U) = f^{-1}(V \cap Z) = f^{-1}(V)$, por ser $f : X \to Z$. Luego es abierto.

```
  · rw [hZ] at hU
    obtain ⟨V, hV⟩ := hU
    rw [← @Set.preimage_val_image_val_eq_self Y Z U, hV.right]
    simp
    apply h
    exact hV.left
```

□

Proposición 3.13. *Sea $f : X \to Y$ una función entre espacios topológicos y sea $\mathcal{B}$ una base de Y. Entonces f es continua si y solo si para cada $U \in \mathcal{B}$ abierto básico, se tiene que $f^{-1}(U)$ es abierto en X.*

Es decir, para la definición de continuidad de una función basta tomar los abiertos básicos.

```
lemma continuous_iff_trueForBasics {X Y : Type} [T : TopologicalSpace
  X]
    [T' : TopologicalSpace Y] (f : X → Y)
    (B : Set (Set Y)) (hB : isTopoBase B) :
    Continuous f ↔ ∀ U ∈ B, IsOpen (f ⁻¹' U)
```

Demostración. La primera implicación es trivial; si la propiedad de continuidad se cumple para cada abierto trivialmente se cumple para los abiertos básicos.

```
    rw [continuous_def]
    constructor; all_goals intro h
    · exact fun U hU ↦ h U (hB.left U hU)
```

($\Longleftarrow$) Sea V abierto en Y y queremos ver que $f^{-1}(V)$ es abierto en X. Como $\mathcal{B}$ es base de Y, existe una familia $\{B_i\}_i \subseteq \mathcal{B}$ de forma que $U = \bigcup_i B_i$. Entonces

$$f^{-1}(U) = f^{-1}\Big(\bigcup_i B_i\Big) = \bigcup_i f^{-1}(B_i),$$

que será abierto cuando cada uno de los componentes de la unión sea abierto. Pero B_i es abierto por pertenecer a una base, y f es continua, luego $f^{-1}(B_i)$ es continuo para cada i.

```
  · intro V hV
    obtain ⟨UB, hUB⟩ := hB.right V hV
    rw [hUB.right, Set.preimage_sUnion]
```

```
    apply isOpen_biUnion
    intro A hA
    apply h
    exact (hUB.left hA)
```

□

Estos dos últimos resultados se pueden combinar, de manera que para demostrar que $f : X \to Z \subseteq Y$ es continua, basta con demostrar la condición para los abiertos básicos de Y. Este resultado, que he demostrado en Lean y llamado `continuousInSubspace_iff_trueForBase`, es uno de los que utilizaremos para demostrar la continuidad en el lema de Urysohn.

3.5 Separación

No cualquier topología sobre un conjunto refleja adecuadamente las propiedades de dicho conjunto. Por ejemplo, la topología trivial no permite diferenciar elementos del espacio, por lo que bajo esta topología no es posible diferenciar unos conjuntos de otros o incluso de un único punto.

Para profundizar en el estudio de la topología, se introducen ciertas condiciones que garantizan que los puntos del espacio puedan distinguirse de alguna forma mediante abiertos. Estas condiciones se conocen como axiomas de separación.

Por ejemplo, un espacio es de Hausdorff si dados dos puntos distintos, existen abiertos disjuntos que contienen a cada uno de ellos. Esta condición es cierta para cualquier espacio métrico, y garantiza ciertas propiedades buenas, como la unicidad de los límites.

En este trabajo nos centraremos, en particular, en los espacios normales.

3.5.1 Espacios normales

Los espacios normales permiten separar no solo puntos, sino conjuntos cerrados disjuntos mediante abiertos disjuntos. Esta propiedad es más exigente, pero también más potente.

Uno de las propiedades más importantes de los espacios normales es que nos permiten distinguir entre conjuntos cerrados separándolos mediante funciones continuas, lo que se conoce como **lema de Urysohn**. La formalización de este resultado es uno de los objetivos principales de este trabajo.

Definición 3.10. *Sea X un espacio topológico. Diremos que X es un espacio* normal *si para cada par de cerrados disjuntos $C, D \subseteq X$ existen abiertos*

disjuntos U y V en X tales que separan C y D, es decir, $C \subseteq U$ y $D \subseteq V$[16].

```
def NormalSpace {X : Type} (T : TopologicalSpace X) : Prop :=
  ∀ C : Set X, ∀ D : Set X,
  IsClosed C → IsClosed D → C ∩ D = ∅ →
  ∃ U : Set X, ∃ V : Set X,
    IsOpen U ∧ IsOpen V ∧ C ⊆ U ∧ D ⊆ V ∧ U ∩ V = ∅
```

Ahora queremos dar una caracterización para este tipo de espacios, que nos facilitará el trabajo más adelante.

Proposición 3.14 (**Caracterización de espacios normales**). *Sea X un espacio topológico. X es normal si y sólo si para cada abierto U y cada cerrado C de X tales que $C \subseteq U$, existe un abierto $V \subset X$ de forma que $C \subseteq V \subseteq \overline{V} \subseteq U$.*

```
lemma characterization_of_normal {X : Type} (T : TopologicalSpace X) :
    NormalTopoSpace T ↔
    ∀ U : Set X, ∀ C : Set X, IsOpen U → IsClosed C → C ⊆ U →
      ∃ V : Set X, IsOpen V ∧ C ⊆ V ∧ (Closure V) ⊆ U := by
```

Demostración. Veamos cada implicación por separado.

```
  rw [normal_space_def]
  constructor
```

($\Longrightarrow$) Supongamos que X es un espacio normal (`hT`) y sean U un abierto (`hU`) y C un cerrado (`hC`) tales que $C \subseteq U$ (`hCU`).

```
  · intro hT U C hU hC hCU
```

Puesto que X es normal, por la definición, para C y U^c cerrados en X obtenemos V_1 y V_2 abiertos (`V1_open`, `V2_open`) disjuntos (`hV`) tales que $C \subseteq V_1$ (`hCV`) y $U^c \subseteq V_2$ (`hUV`).

```
    obtain ⟨V1, V2, V1_open, V2_open, hCV, hUV, hV⟩ :=
      hT C Uᶜ
      hC
      (by exact isClosed_compl_iff.mpr hU)
      (by rw [ABdisjoint_iff_AsubsBc, compl_compl]; exact hCU)
```

Por supuesto, en Lean tenemos que especificar por qué U^c es cerrado y por qué $U^c \subseteq V_2$. Tomamos como V el V_1 obtenido de esta forma. Ya sabemos que V_1 es abierto y que $C \subseteq V_1$, luego sólo falta demostrar que $\overline{V_1} \subseteq U$.

[16] En Lean, la definición `NormalSpace` es ligeramente distinta, pero utiliza objetos que nosotros no hemos usado. En su lugar, utilizamos esta y una demostración de que son equivalentes, a la que he llamado `normal_space_def`.

```
    use V1
    constructor
    · exact V1_open
    constructor
    · exact hCV
```

Sabemos que $U^c \subseteq V_2$ (`hUV`), luego $V_2^c \subseteq U$. Basta ver que $\overline{V_1} \subseteq V_2^c$.

Pero $V_1 \cap V_2 = \emptyset \implies \overline{V_1} \cap V_2 = \emptyset$, por ser V_2 abierto. Luego $\overline{V_1} \subseteq V_2^c$.

```
    · trans V2ᶜ; swap
      · exact Set.compl_subset_comm.mp hUV -- V2ᶜ ⊆ U
      · apply Disjoint.closure_left at hV
        specialize hV V2_open -- hV : Disjoint (closure V1) V2
        exact Set.subset_compl_iff_disjoint_right.mpr hV
          -- closure V1 ⊆ V2ᶜ
```

($\Longleftarrow$) Procedemos de manera similar. Sean C_1 y C_2 cerrados (`C1_closed`, `C2_closed`) disjuntos (`hC`). Podemos aplicar la hipótesis (`h`) al abierto C_1^c y al cerrado C_2 para obtener obtener un abierto V (`V_open`) de manera que $C_2 \subseteq V \subseteq \overline{V} \subseteq C_1^c$ (`hV`).

```
  · intro h C1 C2 C1_closed C2_closed hC
    obtain ⟨V, V_open, hV⟩ :=
      h C1ᶜ C2
      (by exact IsClosed.isOpen_compl) C2_closed
      (by rw [← ABdisjoint_iff_AsubsBc, Set.inter_comm C2 C1];
        exact hC)
```

Ahora tomamos los abiertos $U_1 = \overline{V}^c$ y $U_2 = V$. Queremos ver que cumplen la condición de normalidad para C_1 y C_2, es decir:

```
IsOpen (Closure V)ᶜ ∧ IsOpen V ∧ C1 ⊆ (Closure V)ᶜ ∧ C2 ⊆ V ∧
  (Closure V)ᶜ ∩ V = ∅
```

En efecto, ambos son abiertos ($\overline{V}^c$ por ser el complementario de una clausura y V por construcción).

```
  constructor
  · apply isOpen_compl_iff.mpr
    exact closure_is_closed V
  constructor
  · exact V_open
```

Además, $C_1 \subseteq \overline{V}^c$ es equivalente a $\overline{V} \subseteq C_1^c$, que es cierto por construcción de V, igual que $C_2 \subseteq V$.

```
    constructor
    · apply Set.subset_compl_comm.mp
      exact hV.right
    constructor
    · exact hV.left
```

Por último, se tiene

$$\overline{V}^c \cap V = \emptyset \iff V \cap \overline{V}^c = \emptyset \iff V \subseteq \overline{V}^{cc} \iff V \subseteq \overline{V},$$

que es cierto por las propiedades de la adherencia.

```
    · rw [Set.inter_comm]
      rw [ABdisjoint_iff_AsubsBc]
      simp
      exact set_inside_closure V
```

□

4 El Lema de Urysohn

El lema de Urysohn es uno de los resultados centrales de la topología general. Su importancia radica en que permite caracterizar la normalidad de un espacio topológico en términos de funciones continuas.

Mientras que en los axiomas de separación clásicos se exige la existencia de abiertos disjuntos para separar puntos o conjuntos, existen versiones que requieren separar mediante funciones continuas. Estas versiones son más restrictivas y, por tanto, definen propiedades más fuertes. Sin embargo, el lema de Urysohn muestra que en el caso de los espacios normales, esto no ocurre, sino que separar conjuntos cerrados mediante funciones continuas es equivalente a hacerlo mediante abiertos.

Este resultado tiene aplicaciones clave, como su uso en la demostración del teorema de metrización de Urysohn, que proporciona condiciones suficientes para que un espacio sea metrizable. Además, la demostración del lema es de gran interés por sí misma, especialmente para nuestros propósitos, ya que se basa en una construcción explícita y no trivial de la función buscada. Es precisamente la construcción de esta función a la que dedicaremos buena parte de esta sección.

Teorema 4.1 (**Lema de Urysohn**). *[16, p. 102, 15.6. Urysohn's Lemma] Sea $(X, \mathcal{T})$ un espacio topológico. X es un espacio normal si y solo si para cada par de conjuntos cerrados disjuntos C y D en X, existe una función $f : X \to [0, 1]$ de manera que $f(C) = \{0\}$ y $f(D) = \{1\}$.*

Para la fomalización en Lean, pediremos que los cerrados C y D sean no vacíos. Obviamente, si uno de los dos es vacío, basta tomar la función continua $f(x) \equiv 1$, pero podemos descartar estos casos triviales.

```
lemma Urysohn {X : Type} {Y : Set ℝ} (T : TopologicalSpace X)
    [T' : TopologicalSpace ℝ] (hT' : T' = UsualTopology)
    {R : TopologicalSpace Y} {hY : Y = Set.Icc 0 1}
    {hR : R = TopoSubspace T' Y} :
    NormalSpace X ↔
      ∀ C1 : Set X, ∀ C2 : Set X, C1 ≠ ∅ → C2 ≠ ∅ →
      IsClosed C1 → IsClosed C2 → Disjoint C1 C2 →
      ∃ f : X → Y, Continuous f ∧
        f '' C1 = ({⟨0, by simp [hY]⟩} : Set Y) ∧
        f '' C2 = ({⟨1, by simp [hY]⟩} : Set Y) := by sorry
```

4.1 El recíproco

Veamos primero la demostración del recíproco, que es más sencilla.

Demostración. Supongamos que cualquier par de cerrados disjuntos de X se pueden separar mediante una función continua y veamos que entonces X es un espacio normal. Sean C_1 y C_2 cerrados disjuntos en X.

```
· intro h
  rw [normal_space_def]
  intro C1 C2 hC1 hC2 hinter
```

Como en la definición de normal no pedimos que los cerrados sean no vacíos, tenemos que diferenciar estos casos. Sin embargo, estos casos son triviales porque basta tomar el conjunto vacío para recubrir el vacío y X para recubrir el otro conjunto. En Lean hay que ser rigurosos con este paso, pero aquí lo obviaremos por simplicidad.

Supongamos entonces que C_1 y C_2 son no vacíos. Por hipótesis, existe una función continua $f : X \to [0, 1]$ de forma que $f(C_1) = \{0\}$ y $f(C_2) = \{1\}$.

```
  obtain ⟨f, hf, hfC1, hfC2⟩ := h C1 C2 hC1nempty hC2nempty hC1 hC2
  hinter
```

Consideremos entonces los conjuntos $U_1 = f^{-1}([0, \frac{1}{2}))$ y $U_2 = f^{-1}((\frac{1}{2}, 1])$. Queremos ver que son los abiertos que necesitamos de la definición de normal, es decir, que son abiertos en X, que $C_i \subseteq U_i$ y que son disjuntos.

```
  use f ⁻¹' ({y | (y : ℝ) ∈ Set.Ico 0 (1 / 2)})
  use f ⁻¹' ({y | (y : ℝ) ∈ Set.Ioc (1 / 2) 1})
```

Para ver que U_1 es abierto, utilizamos que f es continua. Basta ver que $[0, \frac{1}{2})$ es abierto en $[0, 1]$. Pero ya vimos que los intervalos de la forma $[0, b)$ son abiertos en $[0, 1]$, así que basta aplicar esta propiedad. Análogo para U_2.

```
  · apply hf -- aplicar def. de f continua
    apply ico_open_in_Icc01 -- [0, b) es abierto en [0, 1]
    · exact hY -- estamos en [0, 1]
    · exact hR -- estamos en la top. relativa
    · norm_num -- 0 < 1/2 < 1
```

Para ver que $C_1 \subseteq U_1$, basta ver que $f(C_1) \subseteq [0, \frac{1}{2})$, que es trivial. Análogo para U_2.

```
  · rw [← Set.image_subset_iff, hfC1] -- {0} ⊆ [0, 1/2) ?
    simp
```

Para ver que son disjuntos, vemos que $[0, \frac{1}{2})$ y $(\frac{1}{2}, 1]$ son disjuntos. Obviamente lo son, pero para Lean es un poco más complicado, así que procedemos por reducción al absurdo para poder simplificar las expresiones. Finalmente llegamos a que no existe un x con $x < 1/2$ y $x > 1/2$.

```
· apply Disjoint.preimage
  by_contra c
  rw [Set.disjoint_iff_inter_eq_empty, ← ne_eq,
    ← Set.nonempty_iff_ne_empty] at c
  obtain ⟨x, hxu, hxv⟩ := c
  simp at hxu hxv
  linarith
```

□

La otra implicación es mucho más compleja, especialmente en su formalización en Lean, como veremos a continuación.

4.2 Esquema de la demostración

Para la demostración de la primera implicación, he seguido la estrategia clásica basada en construir una familia de abiertos indexada por racionales. Aunque existen variantes que emplean subconjuntos particulares como los racionales diádicos, aquí hemos optado por una construcción sobre $\mathbb{Q} \cap [0,1]$, siguiendo versiones habituales del lema presentes en la literatura. Aún así, el esquema que se presenta a continuación es también el resultado del proceso de demostración en Lean, que ha ido tomando forma a medida que se resolvían los distintos obstáculos que se han ido presentando.

Supongamos que X es un espacio normal, y sean C_1 y C_2 dos cerrados disjuntos no vacíos de X.

Consideremos $U_1 = X$ abierto. Consideremos el cerrado C_1 el abierto C_2^c y aplicamos la caracterización de espacios normales (3.14), obteniendo otro abierto U_0 de manera que

$$C_1 \subseteq U_0 \subseteq \overline{U_0} \subseteq C_2^c \subseteq U_1 = X$$

Podemos hacer lo mismo para $\overline{U_0}$ cerrado y C_2^c abierto, obteniendo, por ejemplo, $U_{\frac{1}{2}}$ de forma que

$$C_1 \subseteq U_0 \subseteq \overline{U_0} \subseteq U_{\frac{1}{2}} \subseteq \overline{U_{\frac{1}{2}}} \subseteq C_2^c \subseteq U_1$$

Reiterando este proceso, vamos a construir una sucesión de abiertos sobre $\mathbb{Q} \cap [0,1]$, $\{U_p | p \in \mathbb{Q} \cap [0,1]\}$, de manera que

$$\forall p, q \in \mathbb{Q}, p < q \implies \overline{U_p} \subseteq U_q \qquad (\star)$$

Una vez tenemos esta sucesión, la extenderemos a todo $\mathbb{Q}$, obteniendo lo que en Lean será una función $G : \mathbb{Q} \to \mathcal{P}(X)$. Después, definiremos otra función F sobre X que a cada $x \in X$ le hace corresponder el conjunto

$$F(x) = \{p \in \mathbb{Q} \mid x \in G(p)\}$$

Por último, tomaremos la función $f : X \to [0,1]$ definida por

$$f(x) = \inf F(x)$$

Esta función será la que utilicemos. Tendremos que demostrar que efectivamente toma valores en $[0,1]$, que es continua y que separa nuestros conjuntos cerrados.

Sin embargo, una vez construidas estas funciones, este último paso es relativamente sencillo. La principal dificultad a la hora de formalizar esta demostración ha sido construir función G y demostrar sus propiedades.

Como se puede apreciar en las primeras iteraciones de la construcción de cada U_q, esta sucesión se construye por inducción. Para poder hacer inducción sobre los racionales, nos basamos en que son numerables, y, en particular, en que $\mathbb{Q} \cap [0,1]$ lo es. Vamos a encontrar una función $f : \mathbb{N} \to \mathbb{Q} \cap [0,1]$ biyectiva (invertible), de manera que $f(0) = 1$ y $f(1) = 0$. Esto nos servirá para construir cada U_q.

Después, para demostrar que efectivamente se cumple la condición $(\star)$, necesitaremos utilizar inducción sobre dos variables. Para ello, utilizaremos que el orden lexicográfico de $(\mathbb{N} \times \mathbb{N})$, definido por $(n,m) < (n',m') \iff n < n' \vee (n = n' \wedge m < m')$, es una relación bien fundada, y por tanto admite inducción sobre pares de naturales.

La principal dificultad de esta construcción ha sido encontrar los objetos adecuados para poder formalizar las ideas de la demostración, como la numerabilidad de los racionales o el orden lexicográfico, y aprender la forma correcta en la que trabajar con ciertas estructuras en Lean, como definir funciones recursivas con una recursión no usual.

4.3 Construcción de la sucesión de abiertos

En primer lugar construiremos una sucesión de abiertos sobre $\mathbb{Q} \cap [0,1]$, y después la extenderemos al resto de los racionales. Sea $Q = \mathbb{Q} \cap [0,1]$. Para construir la sucesión $\{U_p \mid p \in Q\}$, o, lo que es lo mismo, la función $G : Q \to \mathcal{P}(X)$, $G(p) = U_p$, vamos a proceder de la siguiente forma.

Como Q es numerable, consideramos la numeración $Q = \{p_k \mid k \in \mathbb{N}\}$, y suponemos por simplicidad que $p_0 = 1$ y que $p_1 = 0$. Vamos a construir los abiertos con la condición $(\star)$ por inducción completa sobre k.

Tomamos entonces, como base de la inducción,

- $U_1 = U_{p_0} = X$.
- $U_0 = U_{p_1}$ el abierto obtenido al aplicar la caracterización de espacios normales (3.14) al cerrado C_1 y el abierto C_2^c.

Trivialmente se tiene que $\overline{U_0} \subseteq U_1 = X$ (con $0 < 1$), luego se satisface $(\star)$.

Ahora, para el caso inductivo, supongamos que hemos definido U_{p_k} para cada $k = 0, 1, ..., n$ satisfaciendo $(\star)$, y ahora queremos construir $U_{p_{n+1}}$.

Notar que el conjunto $\{p_0, p_1, ..., p_n\} \subset Q$ no está necesariamente ordenado. Sin embargo, es un conjunto finito de números racionales, por tanto, podemos encontrar unos elementos p_r y p_s de manera que p_r es el predecesor inmediato de p_{n+1} y p_s es el sucesor inmediato de p_{n+1}. Es decir, se tiene

$$p_r < p_{n+1} < p_s$$

y no existe $k \leq n$ de forma que $p_r < p_k < p_{n+1}$ ni $p_{n+1} < p_k < p_s$.

Puesto que $p_r < p_s$, por la hipótesis de inducción completa se tiene que ambos son abiertos y que $\overline{U_{p_r}} \subseteq U_{p_s}$ $(\star)$. Aplicamos de nuevo la caracterización de espacios normales (3.14), para encontrar un nuevo abierto $U = U_{p_{n+1}}$ tal que

$$\overline{U_{p_r}} \subseteq U \subseteq \overline{U} \subseteq U_{p_s}$$

Con lo que concluye la inducción.

Veamos ahora como se traduce esto en Lean.

4.3.1 Numerar los racionales

Los racionales son numerables, es decir existe una biyección entre $\mathbb{N}$ y $\mathbb{Q}$. En particular, necesitamos una función $f : \mathbb{N} \to Q$ donde $Q = \mathbb{Q} \cap [0, 1]$, de forma que f sea biyectiva, $f(0) = 1$ y $f(1) = 0$. Es decir, la función que nos lleva cada $k \in \mathbb{N}$ a p_k.

```
lemma hf : ∃ f : ℕ → Q,
    (f.Bijective ∧
    f 0 = ⟨1, Q1⟩ ∧
    f 1 = ⟨0, Q0⟩) := by sorry
```

Para demostrar la existencia de tal función necesitamos una serie de resultados previos.

En primer lugar, la numerabilidad de los racionales ya está demostrada en Mathlib como `Rat.instDenumerable`. Para poder extraer una función biyectiva de este resultado, he escrito el siguiente lema:

```
lemma bijective_nat_rat : ∃ f : ℕ → ℚ, f.Bijective  := by
  have f := (Rat.instDenumerable.eqv).symm
  use f
  exact f.bijective
```

Evidentemente, por la independencia de las demostraciones de Lean, no podremos evaluar esta función de manera explícita. Pero tenemos la información que necesitamos de ella.

Ahora, quiero demostrar que existe una función biyectiva de $\mathbb{N}$ en Q. Como ya tenemos una función biyectiva de $\mathbb{N}$ en $\mathbb{Q}$, la idea es componerla con una biyección de $\mathbb{Q}$ en Q.

Para demostrar que esta biyección existe, basta demostrar que $\mathbb{Q}$ y Q tienen el mismo cardinal (`Cardinal.eq`). Pero, de hecho, cualquier subconjunto de $\mathbb{Q}$ no finito tiene cardinal $\aleph_0$ (demostrado en `non_finite_rat_set_cardinal_aleph0`). Basta demostrar que Q no es finito (queda demostrado en `Q_not_finite`).

```
lemma non_finite_rat_set_cardinal_aleph0 (A : Set ℚ) (hA : ¬
  A.Finite) :
    Cardinal.mk ↑A = Cardinal.aleph0 := by sorry
```

Por último, cualquier permutación de dos valores de una función preserva la biyectividad (demostrado en `permute_f_bijectivity`). Por tanto, podemos forzar que $f(0) = 1$ y $f(1) = 0$.

```
def permute_f {X Y : Type} [DecidableEq X]
  (f : X → Y) (a b : X) : X → Y := fun x ↦
    if x = a then f b
    else if x = b then f a
    else f x

lemma permute_f_bijectivity {X Y : Type} [DecidableEq X]
    {f : X → Y} (a b : X) (h : f.Bijective) :
    (permute_f f a b).Bijective := by sorry
```

Una vez demostrado `hf`, podemos definir f mediante `Classical.choose` y empezar a trabajar con ella, aunque no la conozcamos explícitamente.

```
noncomputable def f : ℕ → Q := Classical.choose hf
```

Por ejemplo, podemos probar que tiene inversa.

```
lemma f_has_inverse :  ∃ g, Function.LeftInverse g f ∧
    Function.RightInverse g f := by
  rw [← Function.bijective_iff_has_inverse]
  exact f_prop.left
```

4.3.2 Encontrar el sucesor y predecesor inmediato

Ahora tenemos cada p_k definido en Lean como $f(k)$ para cada $k \in \mathbb{N}$. Para poder definir cada abierto U_{p_k}, necesitamos ser capaces de encontrar para cada conjunto $\{p_0, p_1, \ldots, p_{n-1}\}$, el predecesor inmediato p_r y el sucesor inmediato p_s de p_n.

De nuevo, en Lean esto se codifica como funciones; queremos encontrar una función $r : \mathbb{N} \to \mathbb{N}$ que, para cada $n > 1$, devuelva el predecesor inmediato de $f(n)$, de entre $\{f(k) \mid k < n\}$, y lo mismo para una función $s : \mathbb{N} \to \mathbb{N}$ que encuentre el sucesor inmediato. Sin embargo, la existencia de tales funciones no es trivial.

Lema 4.1. *Sea $n > 1$. Entonces existe un $r_n < n$ de forma que $f(r_n) < f(n)$, y si $k < n$ es tal que $f(k) < f(n)$ entonces $f(k) \leq f(r_n)$.*

```
lemma exists_r (n : ℕ) (hn : n > 1) : ∃ r ∈ Finset.range n,
  ((f r < f n) ∧
  (∀ m ∈ Finset.range n, f m < f n → f m ≤ f r)) := by sorry
```

Demostración. Sea $n > 1$. Consideremos el conjunto

$$R = \{m : \mathbb{N} \mid m < n \wedge f(m) < f(n)\}$$

```
  let R : Finset ℕ := (Finset.range n).filter (fun m ↦ f m < f n)
```

R es un conjunto finito no vacío, pues $1 \in R$

```
  have hR : R.Nonempty
  · use 1; sorry
```

Tomamos el conjunto $f(R)$, que también es un conjunto finito y no vacío, por serlo R, luego tiene máximo. Tomamos el argumento máximo de $f(R)$, $r_n = \arg\max\{f(m) \mid m \in R\}$, y veamos que satisface las condiciones que pedimos.

```
  let fR : Finset ℚ := R.image f
  have hfR : fR.Nonempty := (Finset.image_nonempty).mpr hR
  let fr := Finset.max' fR ((Finset.image_nonempty).mpr hR)
  obtain ⟨r, hr⟩ := Finset.mem_image.mp
    (by exact Finset.max'_mem fR hfR)
  use r
```

Se tiene que $r \in R$, luego $r < n$ y $f(r) < f(n)$. Sea entonces un $k < n$ con $f(k) < f(n)$. Por construcción, $k \in R$ y por tanto $f(k) \in f(R)$. Como r es el argumento máximo de $f(R)$, $f(r)$ es el máximo de $f(R)$ y por tanto $f(k) \leq f(R)$, como queríamos. □

La demostración completa está en el repositorio, así como el análogo para s.

Garantizada la existencia de estas funciones, podemos tomar ahora las funciones r y s y empezar a trabajar con ellas.

```
noncomputable def r : ℕ → ℕ := fun n ↦
  if h : n > 1 then Classical.choose (exists_r n h)
  else 1

noncomputable def s : ℕ → ℕ := fun n ↦
  if h : n > 1 then Classical.choose (exists_s n h)
  else 0
```

Veamos las principales propiedades de estas dos funciones. En primer lugar, tenemos las propiedades básicas de r y s que son simplemente por construcción.

Lema 4.2. *Para cada $n > 1$, se tiene:*

$$\begin{cases} r(n) < n \\ f(r(n)) < f(n) \\ \forall m < n, \text{ si } f(m) < f(n) \text{ entonces } f(m) \leq f(r(n)) \end{cases}$$

```
lemma r_prop (n : ℕ) (hn : n > 1) : (
    (r n ∈ Finset.range n) ∧ (f (r n) < f n) ∧
    (∀ m ∈ Finset.range n, f m < f n → f m ≤ f (r n))
  ) := by sorry
```

El resultado es simétrico para s y recibe el nombre de `s_prop`.

Lema 4.3. *Sea $n > 1$. Entonces o bien $r(n) = 1$ o bien $r(n) > 1$. Es decir, no puede ser $r(n) = 0$. De forma análoga, no puede ser $s(n) = 1$, luego o bien $s(n) = 0$ o bien $s(n) = 1$.*

```
lemma r_options (n : ℕ) (hn : n > 1) : r n = 1 ∨ r n > 1 := by sorry
lemma s_options (n : ℕ) (hn : n > 1) : s n = 0 ∨ s n > 1 := by sorry
```

También se obtiene un resultado `rs_options` que encapsula las cuatro combinaciones posibles de valores para r y s, para facilitar el trabajo con ellas.

Demostración. Si fuera $r(n) = 0$, tendríamos $1 = f(0) = f(r(n)) < f(n)$, lo que es imposible pues f toma valores en $[0, 1]$. □

Lema 4.4. *Sea $n > 1$ y supongamos que $s(n) > 1$ y que $r(n) < s(n)$. Entonces $r(n) = r(s(n))$.*

De manera análoga, si $r(n) > 1$ y $s(n) < r(n)$, entonces $s(n) = s(r(n))$.

```
lemma rn_eq_rsn (n : ℕ) (hn : n > 1) (hsn : s n > 1)
    (h : r n < s n) : r n = r (s n) := by
```

El análogo a este recibe el nombre de `sn_eq_srn`.

Demostración. Demostramos solo el primero y el segundo es parecido. Sea $n > 1$ y supongamos que $s(n) > 1$ y que $r(n) < s(n)$. Queremos ver que $r(n) = r(s(n))$. Por la inyectividad de f, podemos comprobar en su lugar que $f(r(n)) = f(r(s(n)))$. Veamos que se cumplen ambas desigualdades.

```
  apply f_prop.left.left
  apply ge_antisymm
```

(1) Para ver $f(r(s(n))) \leq f(r(n))$, aplicamos la tercera parte de `r_prop`. Luego hay que ver que $r(s(n)) < n$, lo cual sencillo por ser $r(s(n)) < s(n) < n$, y que $f(r(s(n))) < fn$. Esto último es cierto porque si fuera $f(n) < f(r(s(n)))$, como $r(s(n)) < s(n)$ tendríamos $f(s(n)) \leq f(r(s(n)))$, pero eso es imposible porque $f(r(s(n))) < f(s(n))$ por construcción (esta última parte viene dada por un resultado auxiliar `f_rs_prop`).

```
  · -- f (r (s n)) ≤ f (r n)
    apply (r_prop n hn).right.right
    · simp
      trans s n
      · exact List.mem_range.mp (r_prop (s n) hsn).left
      · exact List.mem_range.mp (s_prop n hn).left
    · exact f_rs_prop n hn hsn
```

(2) Para ver que $f(r(n)) \leq f(r(s(n)))$, basta ver que $r(n) < s(n)$ por hipótesis y que $f(r(n)) < f(n) < f(s(n))$ por las propiedades de r y s. Luego aplicando la tercera propiedad básica a $s(n) > 1$ y $r(n) < s(n)$ obtenemos el resultado.

```
  · -- f (r n) ≤ f (r (s n))
    apply (r_prop (s n) hsn).right.right
    · exact List.mem_range.mpr h
    · trans f n
      · exact (r_prop n hn).right.left
      · exact (s_prop n hn).right.left
```

□

4.3.3 Construcción de G

La construcción de $G : \mathbb{Q} \to \mathcal{P}(X)$ es, como hemos explicado, una construcción por inducción. Para empezar, es más fácil construir $G : \mathbb{N} \to \mathcal{P}(X)$, y después

tomar $G \circ f^{-1} : Q \to \mathcal{P}(X)$ donde f es la función que numera Q que habíamos obtenido antes.

Lean admite definiciones inductivas de una manera muy natural. Un ejemplo muy utilizado es la sucesión de Fibonacci: definir $Fib(0) = 0$ y $Fib(1) = 1$, y a partir de ahí, $Fib(n) = Fib(n-1) + Fib(n-2)$ para cada $n > 1$. En Lean, escribimos

```
def Fib : ℕ → ℕ := fun n ↦
  if n = 0 then 0
  else if n = 1 then 1
  else Fib (n-1) + Fib (n-2)
```

En vista de esto, mi primer acercamiento a la construcción de G fue el siguiente:

- Para $n = 0$, definir $G(0) = C_2^c$.
- Para $n = 1$, tomar $G(1)$ el resultado de aplicar la caracterización de espacios normales al abierto C_2^c y el cerrado C_1.
- Para $n > 1$, tomar $G(n)$ el resultado de aplicar la caracterización de espacios normales (3.14) al abierto $G(s(n))$ y el cerrado $\overline{G(r(n))}$.

```
def G {X : Type} [TopologicalSpace X]
  (hT : ∀ (U C : Set X), IsOpen U → IsClosed C → C ⊆ U →
    ∃ V, IsOpen V ∧ C ⊆ V ∧ closure V ⊆ U)
  (C1 C2 : Set X) (hC1 : IsClosed C1)
  (hC2 : IsOpen C2ᶜ) (hC1C2 : C1 ⊆ C2ᶜ)
  : ℕ → Set X := fun n ↦
    if n = 0 then C2ᶜ
    else if n = 1 then Classical.choose (hT C2ᶜ C1 hC2 hC1 hC1C2)
    else
      let U := g hT C1 C2 hC1 hC2 hC1C2 (s n)
      let C := closure (g hT C1 C2 hC1 hC2 hC1C2 (r n))
      Classical.choose (hT U C (by sorry) (by sorry) (by sorry))
```

Donde en los últimos `sorry` habría que demostrar que $G(s(n))$ es abierto, que $\overline{G(r(n))}$ es cerrado y que $\overline{G(r(n))} \subseteq G(s(n))$, para poder aplicar la caracterización de la normalidad (`hT`).

Todo esto nosotros "lo sabemos": es la hipótesis de inducción. Pero a Lean todavía no le hemos dicho nada de eso. ¿Cómo podemos probar algo sobre un objeto que aún no hemos definido, porque necesitamos haberlo probado para poder definirlo?

Para evitar este problema, tomé una estrategia distinta. Primero, defino la noción de *par normal* de la siguiente manera.

Definición 4.1. *Dados $U, C \subseteq X$, decimos que (U, C) es un* par normal *si U es abierto, C es cerrado y $C \subseteq U$. Es decir, si satisfacen las condiciones para poder aplicar la caracterización de espacios normales (3.14).*

```
def normal_pair {X : Type} [TopologicalSpace X]
  : (Set X × Set X) → Prop := fun (U, C) ↦
  (IsOpen U ∧ IsClosed C ∧ C ⊆ U)
```

Ahora, defino una función `from_normality` que toma dos conjuntos cualesquiera de X, y devuelve el resultado de aplicar la caracterización de espacios normales si forman un par normal, y el conjunto vacío en caso contrario.

```
noncomputable def from_normality {X : Type} [T : TopologicalSpace X]
  (hT : ∀ (U C : Set X), IsOpen U → IsClosed C → C ⊆ U → ∃ V,
  IsOpen V ∧ C ⊆ V ∧ closure V ⊆ U)
  : (Set X × Set X) → Set X := fun (U, C) ↦
    if h : normal_pair (U, C) = True then
      Classical.choose (hT U C
        (by simp at h; exact h.left)
        (by simp at h; exact h.right.left)
        (by simp at h; exact h.right.right)
      )
    else ∅
```

Ahora, al construir G, ya no me tengo que preocupar de si produce siempre conjuntos abiertos o no, porque lo puedo definir a partir de esta última función, que está definida para cualquieras dos conjuntos. Una vez definida, puedo demostrar que cada conjunto obtenido es de hecho abierto.

```
def G {X : Type} [T : TopologicalSpace X]
  (hT : ∀ (U C : Set X), IsOpen U → IsClosed C → C ⊆ U →
    ∃ V, IsOpen V ∧ C ⊆ V ∧ closure V ⊆ U)
  (C1 C2 : Set X) : ℕ → Set X
    := fun n ↦
      if n = 0 then C2ᶜ
      else if n = 1 then from_normality hT (C2ᶜ, C1)
      else if n > 1 then
        let U := G hT C1 C2 (s n)
        let C := closure (G hT C1 C2 (r n))
        from_normality hT (U, C)
      else ∅
```

Sin embargo, a Lean esto sigue sin parecerle suficiente, y recibimos este error: `fail to show termination for G`. Esto es, estamos definiendo una función recursiva, G, pero no es evidente que las veces que estamos llamando a G dentro de G constituyan conjuntos ya definidos. Es decir, no es evidente que $r(n) < n$ y $s(n) < n$. Por tanto, al final de la definición tenemos que añadir una demostración de que sí es así.

```
...
    else ∅

  decreasing_by
  · let s_prop := s_prop
    have aux : ∀ n > 1, s n < n
    · intro n hn
      specialize s_prop n hn
      simp at s_prop
      exact s_prop.left
    apply aux
    linarith
  · sorry -- analogo r
```

¡Ya tenemos nuestra función G! Aunque todavía queda mucho trabajo. Veamos que se cumplen las propiedades que queríamos de G.

Lema 4.5. *Para cada $n \in \mathbb{N}$, $G(n)$ es abierto en X.*

```
lemma G_Prop1 {X : Type} [T : TopologicalSpace X]
  (hT : ∀ (U C : Set X), IsOpen U → IsClosed C → C ⊆ U →
    ∃ V, IsOpen V ∧ C ⊆ V ∧ closure V ⊆ U)
  (C1 C2 : Set X) (hC1 : IsClosed C1) (hC2 : IsOpen C2ᶜ)
  (hC1C2 : C1 ⊆ C2ᶜ)
  :
  ∀ n : ℕ, IsOpen (G hT C1 C2 n) := by
```

Demostración. Notemos que la función `from_normality` siempre produce abiertos, porque o bien es el resultado de aplicar 3.14, o bien es el vacío que es abierto (demostrado en `from_normality_open`).

Sea $n \in \mathbb{N}$. Tenemos que distinguir en tres casos, pues existen tres casos en la definición de G.

(1) Si $n = 0$, entonces simplemente por hipótesis $G(0) = C_2^c$ es abierto.

```
  intro n
  cases' Nat.eq_zero_or_pos n with hn hn -- n = 0 ∨ n > 0
  · simp [hn, G] -- si n = 0
    exact { isOpen_compl := hC2 }
```

(2) Si $n = 1$, entonces estamos aplicando la función `from_normality` a C_2^c y C_1. Luego es abierto.

```
  cases' LE.le.eq_or_gt hn with hn hn -- n = 1 ∨ n > 1
  · simp [hn, G] -- n = 1
    exact from_normality_open hT C2ᶜ C1
```

(3) Ahora, si $n > 1$, entonces estamos aplicando la función `from_normality` a $G(s(n))$ y $\overline{G(r(n))}$. Luego es abierto análogamente. □

Para el siguiente resultado vamos a utilizar inducción completa. Yo he definido mi propio principio de inducción completa, demostrado a partir de la inducción completa usual en Lean, por sencillez.

```
theorem my_stronger_induction (n : ℕ) (P Q : ℕ → Prop)
  (hn : P n) (h : ∀ n : ℕ, P n → ((∀ m < n, P m → Q m) → Q n)) :
  (Q n) := by sorry
```

Lema 4.6. *Para cada $n > 1$, se tiene:*

$$\overline{G(r(n))} \subseteq G(n) \subseteq \overline{G(n)} \subseteq G(s(n))$$

```
lemma G_Prop2 {X : Type} [T : TopologicalSpace X]
  (hT : ∀ (U C : Set X), IsOpen U → IsClosed C → C ⊆ U →
    ∃ V, IsOpen V ∧ C ⊆ V ∧ closure V ⊆ U)
  (C1 C2 : Set X) (hC1 : IsClosed C1) (hC2 : IsOpen C2ᶜ)
  (hC1C2 : C1 ⊆ C2ᶜ)
  :
  ∀ n > 1, closure (G hT C1 C2 (r n)) ⊆ (G hT C1 C2 n)
    ∧ closure (G hT C1 C2 n) ⊆ (G hT C1 C2 (s n)) := by
```

Demostración. Procedemos por inducción completa sobre n.

```
  intro n hn
  let P : ℕ → Prop := fun m ↦ m > 1
  apply my_stronger_induction n P
  exact hn -- probar que n cumple P
  intro n hn hi
```

Tenemos la siguiente hipótesis de inducción: para cada $m < n$ con $m > 1$ se tiene que $\overline{G(r(m))} \subseteq G(m) \subseteq \overline{G(m)} \subseteq G(s(m))$.

Queremos ver que entonces lo mismo se tiene para n.

```
hi : ∀ m < n, 1 < m → closure (G hT C1 C2 (r m)) ⊆ G hT C1 C2 m ∧
  closure (G hT C1 C2 m) ⊆ G hT C1 C2 (s m)
```

```
⊢ closure (G hT C1 C2 (r n)) ⊆ G hT C1 C2 n ∧ closure (G hT C1 C2 n)
  ⊆ G hT C1 C2 (s n)
```

Notemos que si $G(n)$ está obtenido mediante 3.14 aplicado a $G(s(n))$ y $\overline{G(r(n))}$, lo anterior se reduce a comprobar que $(G(s(n)), \overline{G(r(n))})$ es un par normal (demostrado en `from_normality_prop2`).

```
have normalpair : normal_pair (G hT C1 C2 (s n), closure (G hT C1 C2
(r n)))
```

Hemos visto que $r(n)$ es o bien $r(n) = 1$ o bien $r(n) > 1$ (`r_options`) y $s(n)$ es o bien $s(n) = 0$ o bien $s(n) > 1$ (`s_options`). Veamos por ejemplo el caso $r(n) = 1, s(n) > 1$. El resto son parecidos y se pueden consultar en el repositorio.

(1) Ver que $G(s(n))$ es abierto es sencillo; ya hemos visto que G siempre devuelve abiertos.

```
...
  · exact G_Prop1 hT C1 C2 hC1 hC2 hC1C2 (s n)
```

(2) Ver que $\overline{G(r(n))}$ es cerrado es sencillo; ya hemos visto que la adherencia de cualquier conjunto es cerrada.

```
...
  · exact isClosed_closure
```

(3) Ahora, para ver que $\overline{G(r(n))} \subseteq G(s(n))$, necesitamos utilizar la hipótesis de inducción.

Sabemos que $s(n) < n$. Luego podemos aplicar la hipótesis de inducción a $s(n)$, obteniendo que

$$\overline{G(r(s(n)))} \subseteq G(s(n)) \subseteq \overline{G(s(n))} \subseteq G(s(s(n)))$$

Pero, puesto que $s(n) > 1$, se tiene que $r(n) = r(s(n))$ (`rn_eq_rsn`), obteniendo $\overline{G(r(n))} \subseteq G(s(n))$.

```
...
  · have hsn := (s_prop n hn).left -- hsn : s n ∈ Finset.range n
    simp at hsn -- hsn : s n < n
    specialize hi (s n) hsn hs -- aplicar la H.I. a s(n)
    rw [rn_eq_rsn n hn hs (by linarith)] -- usar r(n) = r(s(n))
    exact hi.left
```

□

4.3.4 La propiedad ($\star$)

Esta es la propiedad más importante que queremos pedirle a la función G, pues es la que garantiza la continuidad de la función que queremos construir para la demostración del lema de Urysohn, como veremos más adelante.

Lema 4.7. *Sean $n, m \in \mathbb{N}$ con $f(n) < f(m)$. Entonces*

$$\overline{G(n)} \subseteq G(m)$$

```
lemma G_Prop2_ext {X : Type} [T : TopologicalSpace X]
  (hT : ∀ (U C : Set X), IsOpen U → IsClosed C → C ⊆ U →
    ∃ V, IsOpen V ∧ C ⊆ V ∧ closure V ⊆ U)
  (C1 C2 : Set X) (hC1 : IsClosed C1) (hC2 : IsOpen C2ᶜ)
  (hC1C2 : C1 ⊆ C2ᶜ)
  :
  ∀ n m, f n < f m → closure (G hT C1 C2 n) ⊆ G hT C1 C2 m := by
  sorry
```

Para demostrar este resultado vamos a utilizar un principio de inducción más general que la inducción usual sobre los naturales. Siempre que se cuente con una relación bien fundada en un conjunto, se puede describir un principio de inducción sobre este conjunto.

En nuestro caso, vamos a utilizar el orden lexicográfico sobre $\mathbb{N}^2$, definido por $(n, m) < (n', m') \iff n < n' \vee (n = n' \wedge m < m')$. Este ya está definido en Mathlib, y también está demostrado que es una relación bien fundada.

```
def lt_pair : (ℕ × ℕ) → (ℕ × ℕ) → Prop := Prod.Lex (Nat.lt)
  (Nat.lt)
def lt_pair_wfr : WellFoundedRelation (ℕ × ℕ) := Prod.lex
  (Nat.lt_wfRel) (Nat.lt_wfRel)
lemma lt_pair_wf : WellFounded lt_pair := lt_pair_wfr.wf
```

Demostración. Procedemos por tanto por inducción bien fundada sobre el par (n, m)

```
    := by
  ... -- ciertas modificaciones del objetivo
  apply WellFounded.induction lt_pair_wf
  ...
  intro n m
  intro hi hnm
```

Por tanto tenemos la siguiente hipótesis de inducción:

```
  hi : ∀ (a b : ℕ), lt_pair (a, b) (n, m) → f a < f b → closure (G
  hT C1 C2 a) ⊆ G hT C1 C2 b
```

Ahora dividimos entre distintos casos de valores de n y m. Nos centramos en el caso $n > 1, m > 1$, que es el más interesante, y el resto se pueden consultar en el repositorio.

Además, hay tres casos posibles: o bien $f(s(n)) < f(m)$, o bien $f(s(n)) = f(m)$, o bien $f(s(n)) > f(m)$.

(1) Si $f(s(n)) < f(m)$, tenemos por un lado que $\overline{G(s(n))} \subseteq G(m)$, por ser $(s(n), m) < (n, m)$ y la hipótesis de inducción.

Por otro lado, $\overline{G(n)} \subseteq G(s(n))$, por la segunda propiedad de G. Luego tenemos

$$\overline{G(n)} \subseteq G(s(n)) \subseteq \overline{G(s(n))} \subseteq G(m)$$

```
...
  · -- si f (s n) < f m
    trans closure (G hT C1 C2 (s n))
    · trans G hT C1 C2 (s n)
      · exact (G_Prop2 n hn1).right
      · exact subset_closure
    · exact hi (s n) m (by left; exact s_prop.left) h
```

(2) Si $f(s(n)) = f(m)$ es muy fácil, porque entonces por la inyectividad de f se tiene $s(n) = m$, y $\overline{G(n)} \subseteq G(s(n)) = G(m)$ por la segunda propiedad de G.

```
...
  · -- si f (s n) = f m
    apply f_prop.left.left at h
    rw [h]
    exact (G_Prop2 n hn1).right
```

(3) Supongamos ahora que $f(s(n)) > f(m)$. Este es el caso más complicado, y requiere que volvamos a dividir en tres posibles opciones para $f(n) \sim f(r(m))$.

Si $f(n) < f(r(m))$, procedemos de manera parecida a (1), utilizando la hipótesis de inducción para $(n, r(m)) < (n, m)$. Si $f(n) = f(r(m))$, hacemos lo mismo que en (2). Por último, queda demostrar que no se puede dar $f(r(m)) < f(n)$.

En efecto, si $f(s(n)) > f(m)$, se tiene que dar $n < m$, porque si fuera $m < n$ tendríamos $f(n) < f(m) < f(s(n))$, lo cual es imposible por las propiedades de s. Pero entonces si fuera $f(r(m)) < f(n)$ tendríamos $f(r(m)) < f(n) < f(m)$ con $n < m$, que es imposible por las propiedades de r.

```
...
  · -- si f (r m) < f n
    by_contra
    have r_prop := r_prop.right.right n n_lt_m hnm -- f n ≤ f (r m)
    apply not_lt.mpr at r_prop -- ¬f (r m) < f n
    exact r_prop h' -- teniamos h' : f (r m) < f n
```

□

4.3.5 Composición con f^{-1}

Por último, recordemos que la función G que estábamos buscando no es de la forma $\mathbb{N} \to \mathcal{P}(X)$ sino de la forma $\mathbb{Q} \cap [0,1] \to \mathcal{P}(X)$. En particular, vamos a extenderla a $\mathbb{Q} \to \mathcal{P}(X)$ de la siguiente forma:

$$H : \mathbb{Q} \to \mathcal{P}(X), \quad q \mapsto \begin{cases} \emptyset & \text{si } q < 0 \\ (G \circ f^{-1})(q) & \text{si } 0 \leq q \leq 1 \\ X & \text{si } 1 < q \end{cases}$$

```
def H {X : Type} [TopologicalSpace X]
  (hT : ∀ (U C : Set X), IsOpen U → IsClosed C → C ⊆ U →
    ∃ V, IsOpen V ∧ C ⊆ V ∧ closure V ⊆ U)
  (C1 C2 : Set X)
  : ℚ → Set X := fun q ↦
    if q < 0 then ∅
    else if h : 0 ≤ q ∧ q ≤ 1 then G hT C1 C2 (f_inv ⟨q, h⟩)
    else Set.univ
```

Lema 4.8. *Para los cerrados C_1 y C_2 elegidos al principio de la sección, la función H previamente definida satisface las siguientes propiedades:*

1. $H(1) = C_2^c$
2. $H(0)$ *es tal que* $C_1 \subseteq H(0) \subseteq \overline{H(0)} \subseteq C_2^c$
3. *Para cada* $q \in \mathbb{Q}$, $H(q)$ *es abierto en* X.
4. *Para cada* $p, q \in \mathbb{Q}$ *con* $p < q$, *se tiene* $\overline{H(p)} \subseteq H(q)$

Demostración. Todas estas propiedades se deducen casi trivialmente de las de G, utilizando que f es biyectiva y los valores fijos que toma en el 0 y el 1. El único añadido son los valores fuera de $[0,1]$, para los que las últimas propiedades se demuestran fácilmente puesto que $\emptyset$ y X son abiertos, $\overline{\emptyset} = \emptyset \subseteq U$ para cualquier U y $U \subseteq X$ para cualquier U. □

4.4 Construcción de una función continua separadora

Siguiendo con el esquema que habíamos propuesto, construimos ahora la siguiente función

$$\begin{array}{rcccl} F & : & X & \longrightarrow & \mathcal{P}(\mathbb{Q}) \\ & & x & \longmapsto & \{p \in \mathbb{Q} \mid x \in H(p)\} \end{array}$$

```
def F {X : Type} [TopologicalSpace X]
  (hT : ∀ (U C : Set X), IsOpen U → IsClosed C → C ⊆ U →
    ∃ V, IsOpen V ∧ C ⊆ V ∧ closure V ⊆ U)
  (C1 C2 : Set X)
  : X → Set ℚ := fun x : X ↦ {p : ℚ | x ∈ H hT C1 C2 p}
```

Ahora, queremos construir una función de la siguiente forma

$$\begin{array}{rcccl} f & : & X & \longrightarrow & [0,1] \\ & & x & \longmapsto & \text{ínf}\, F(x) \end{array}$$

Sin embargo, tenemos que asegurarnos de que podemos definir esta función, es decir, de que el conjunto $F(x)$ tiene ínfimo para cada $x \in X$.

4.4.1 La función F

Lema 4.9. *Sea $F : X \to \mathcal{P}(\mathbb{Q})$ la función definida anteriormente. Entonces para cada $x \in X$, $F(x)$ es un conjunto no vacío.*

Demostración. Sea $x \in X$. Cualquier $q > 1$ es tal que $q \in F(x)$, pues si $q > 1$, $H(q) = X$, luego $x \in H(q)$. □

```
lemma hF_non_empty (...) :  ∀ x : X, (F hT C1 C2 x).Nonempty := by
  intro x
  use 2 -- por ejemplo 2 > 1
  simp [F, H]
```

Lema 4.10. *Para cada $x \in X$, si $q < 0$ entonces $q \notin F(x)$. Es decir, todos los elementos de $F(x)$ son no negativos.*

Demostración. Sea $x \in X$ y $q < 0$. Entonces $H(q) = \emptyset$, luego obviamente $x \notin H(q)$. □

Llamaremos a este resultado `hFx_non_neg`. Como consecuencia, se tiene:

Lema 4.11. *Para cada $x \in X$, 0 es una cota inferior de $F(x)$.*

```
lemma hFx_has_lb_0 (...)
  : ∀ x : X, 0 ∈ lowerBounds (F hT C1 C2 x) := by sorry
```

Por tanto, dado $x \in X$, puesto que $F(x)$ es un conjunto no vacío y acotado inferiormente, podemos concluir que $F(x)$ tiene un ínfimo **como conjunto real**, es decir, podemos concluir que existe un número $r \in \mathbb{R}$ de forma que $r = \text{ínf}\, F(x)$. Este resultado está incluido en Mathlib:

```
Real.exists_isGLB {s : Set ℝ} (hne : s.Nonempty) (hbdd : BddBelow s) :
    ∃ x, IsGLB s x
```

Podríamos intentar escribir algo así:

```
lemma hFx_has_lb_0 (...)
  : ∀ x : X, ∃ r : ℝ, IsGLB (F hT C1 C2 x) r := by sorry
```

Sin embargo, esto nos da el siguiente error:

```
argument r has type ℝ but is expected to have type ℚ
```

Esto es porque, en Mathlib, `IsGLB` está definido de la siguiente forma:

```
def IsGLB [Preorder α] (s : Set α) : α → Prop :=
  IsGreatest (lowerBounds s)
```

Es decir, sólo está definido para valores de tipo α si $s \subset \alpha$. Como $F(x) \subset \mathbb{Q}$, no podemos decir que $r \in \mathbb{R}$ sea su ínfimo.

Por tanto, necesitamos definir en Lean una función auxiliar $\tilde{F}$ de forma que $\tilde{F}(x) = F(x)$ para cada x, pero visto como subconjunto de $\mathbb{R}$. Lo hacemos de la siguiente manera.

```
def inclQR : ℚ → ℝ := fun q ↦ q

def F_Real (...) : X → Set ℝ :=
  fun x ↦ inclQR '' (F hT C1 C2 x)
```

Esta nueva función $\tilde{F}$ es una función con las mismas propiedades que F, es decir, para cada $x \in X$, $\tilde{F}(x)$ es un conjunto no vacío con 0 como cota inferior. Esto se prueba de manera inmediata utilizando las propiedades de F, aunque es necesario tener cuidado con la inferencia de tipos de Lean en algunos casos.

Pero ahora $\tilde{F}$ devuelve subconjuntos de $\mathbb{R}$, luego podemos asegurar que tiene un ínfimo en $\mathbb{R}$.

```
lemma F_Real_has_inf (...)
    : ∀ x : X, ∃ r : ℝ, IsGLB (F_Real hT C1 C2 x) r := by
  intro x
  apply Real.exists_isGLB
  exact F_Real_Nonempty hT C1 C2 x
  exact F_Real_BddBelow hT C1 C2 x
```

4.4.2 La función f

Una vez hemos asegurado que $\tilde{F}$ tiene ínfimo, podemos intentar definir nuestra función f como queríamos. Antes de eso, hagamos la siguiente anotación.

Nosotros queremos encontrar una función $f : X \to [0,1]$. Sin embargo, sólo hemos garantizado que ínf $\tilde{F}(x) \in \mathbb{R}$; no necesariamente ínf $\tilde{F}(x) \in [0,1]$.

Evidentemente, esto último es cierto, porque 0 es cota inferior, luego el ínfimo será al menos 0, y $\forall q > 1, q \in F(x)$, luego el ínfimo no podrá ser mayor que 1.

Sin embargo, por facilidad a la hora de trabajar con Lean, definiremos primero una función que sea simplemente

$$\begin{array}{rccc} k & : & X & \longrightarrow & \mathbb{R} \\ & & x & \longmapsto & \text{ínf } \tilde{F}(x) \end{array}$$

y probamos que en efecto $k(x) \in [0,1]$ para cada $x \in X$. Más tarde, utilizaremos la función $f(x) = k(x)$ teniendo cuidado en especificar que $k(x) \in [0,1]$.

```
noncomputable def k (...) : X → ℝ :=
  fun x ↦ Classical.choose (F_Real_has_inf hT C1 C2 x)
```

Obtenemos la siguiente propiedad, fruto de aplicar `Classical.choose_spec` a k.

```
lemma k_prop (...)
  : ∀ x, IsGLB (F_Real hT C1 C2 x) (k hT C1 C2 x) := by sorry
```

Lema 4.12. *Para cada $x \in X$, la función $k : X \to \mathbb{R}$ que acabamos de definir satisface $k(x) \in [0,1]$.*

```
lemma k_in_01 (...)
  : ∀ x : X, (k hT C1 C2 x) ∈ Set.Icc 0 1 := by
```

Demostración. La prueba es sencilla y ya la hemos explicado, pero en Lean necesitamos dar una especificación completa, así que la explicamos.

Sea $x \in X$. Recordemos que $k(x)$ es cota inferior de $\tilde{F}(x)$ (`klb`) y es la mayor de ellas (`kglb`).

```
  intro x
  have ⟨klb, kglb⟩ := k_prop hT C1 C2 x
  constructor
```

Ver que $k(x) \geq 0$ es fácil, porque ya hemos visto que 0 es cota superior, y $k(x)$ es la mayor.

```
    · exact kglb (F_Real_0_is_LB hT C1 C2 x)
```

Ahora, para ver que $k(x) \leq 1$, procedemos por reducción al absurdo. Supongamos que $k(x) > 1$. Entonces existe un número racional q de forma que $1 < q < k(x)$. Como $q > 1$, sabemos que $q \in \tilde{F}(x)$. Pero en ese caso, como $k(x)$ es cota inferior, $k(x) \leq q$, lo que es contradictorio.

```
    · by_contra c
      simp at c
      obtain ⟨q, hq1, hqk⟩ := exists_rat_btwn c -- obtener q
      have hq := F_Real_1inf hT C1 C2 x q (by exact_mod_cast hq1)
        -- hq : q ∈ F_Real x
      apply klb at hq -- hq : k x ≤ q
      apply not_lt.mpr at hq -- hq : ¬ q < k x
      exact hq hqk -- contradiccion
```

□

Por último, antes de pasar a la demostración del lema de Urysohn finalmente, damos varias propiedades de k que necesitaremos. Recordemos que H era $G \circ f^{-1}$ extendida a $\mathbb{Q}$.

Lema 4.13. *Para cada $p \in \mathbb{Q}$ y cada $x \in X$, si $x \in \overline{H(p)}$ entonces $k(x) \leq p$.*

```
lemma k_claim1 (...)
  : ∀ p : ℚ, ∀ x : X, x ∈ closure (H hT C1 C2 p) →
  (k hT C1 C2 x) ≤ p := by sorry
```

Demostración. Sean $p \in \mathbb{Q}$ y $x \in X$ con $x \in \overline{H(p)}$. Supongamos, por reducción al absurdo, que $k(x) > p$. Entonces existe un racional $q \in \mathbb{Q}$ de forma que $p < q < k(x)$.

```
    intro p x hx -- hx : x ∈ closure (H hT C1 C2 p)
    by_contra c
    simp at c -- c : p < k hT C1 C2 x
    have ⟨q, hq⟩ := exists_rat_btwn c
```

Como $p < q$ y $x \in \overline{H(p)}$, por $(\star)$ se tiene que $x \in H(q)$.

```
    apply H_isOrdered hT C1 C2 hC1 hC2 hC1C2 p q
      (by exact_mod_cast hq.left)
      at hx -- x ∈ H hT C1 C2 q
```

Ahora, $x \in H(q)$ quiere decir que $q \in \tilde{F}(x)$. Pero $k(x)$ es cota superior de $\tilde{F}(x)$, luego $k(x) < q$, lo que contradice $q < k(x)$.

```
  have aux : inclQR q ∈ F_Real hT C1 C2 x := by sorry
  have ⟨klb, _⟩ := k_prop hT C1 C2 x
  apply klb at aux -- aux : k x ≤ q
  apply not_lt.mpr at aux -- aux : ¬ q < k x
  exact aux hq.right -- contradiccion
```

□

Lema 4.14. *Para cada $p \in \mathbb{Q}$ y cada $x \in X$, si $x \notin H(p)$ entonces $k(x) \geq p$.*

```
lemma k_claim2 (...)
  : ∀ p : ℚ, ∀ x : X, x ∉ (H hT C1 C2 p) →
  (k hT C1 C2 x) ≥ p := by sorry
```

Demostración. La demostración es muy parecida a la anterior y puede encontrarse en el repositorio. □

4.4.3 Propiedades sobre los cerrados C_1 y C_2

Por último, veamos cómo se comportan la función k en los cerrados que queremos separar. Esto nos dejará el camino completamente allanado para completar la demostración del lema de Urysohn. Vamos a ver las demostraciones para C_1, y para C_2 son similares y se pueden comprobar en el repositorio.

Queremos llegar a que $k(C_1) = \{0\}$. Esta propiedad se construye paso a paso, utilizando propiedades de las funciones en las que se apoya.

Lema 4.15. *Consideremos $F : X \to \mathcal{P}(\mathbb{Q})$. Para cada $x \in C_1$, se tiene que $F(x) = \{q \in \mathbb{Q} \mid q \geq 0\}$.*

```
lemma F_at_C1 (...)
    : ∀ x : X, x ∈ C1 → F hT C1 C2 x = {q : ℚ | q ≥ 0} := by
```

Demostración. Sea $x \in C_1$. Ver que $F(x) \subseteq \{q \in \mathbb{Q} \mid q \geq 0\}$ es sencillo, pues ya hemos visto que ningún valor de $F(x)$ es negativo (`hFx_non_neg`). Ahora, sea $q \geq 0$ y veamos que $q \in F(x)$, es decir, que $x \in H(q)$.

Si $q > 0$, por la propiedad $(\star)$ de H, $\overline{H(0)} \subseteq H(q)$. Y por construcción de H se tiene $C_1 \subseteq H(0)$. Luego $x \in H(q)$. Si $q = 0$, como $C_1 \subseteq H(0)$, $x \in H(q)$.

```
    · cases' Decidable.lt_or_eq_of_le hq with hq hq
      · apply H_isOrdered hT C1 C2 hC1 hC2 hC1C2 0 q hq -- si q > 0
        apply subset_closure
        exact H_C1_in_H0 hT C1 C2 hC1 hC2 hC1C2 hx
```

```
      · rw [← hq] -- si q = 0
        exact H_C1_in_H0 hT C1 C2 hC1 hC2 hC1C2 hx
```

□

Lema 4.16. *Consideremos $F : X \to \mathcal{P}(\mathbb{Q})$. Para cada $x \in C_1$, se tiene que* ínf $F(x) = 0$.

```
lemma F_0_GLB_in_C1 (...)
    : ∀ x : X, x ∈ C1 → IsGLB (F hT C1 C2 x) 0 := by sorry
```

Una vez sabemos que $F(x) = \{q \in \mathbb{Q} \mid q \geq 0\}$, ver que el ínfimo de este conjunto es 0 es directo. Notemos que aquí sí podemos definir el ínfimo sobre F porque $0 \in \mathbb{Q}$. Ahora podemos afirmar lo mismo para $\tilde{F}$, simplemente teniendo cuidado con el tipo de $F(x)$ y de 0.

Lema 4.17. *Consideremos $\tilde{F} : X \to \mathcal{P}(\mathbb{R})$. Para cada $x \in C_1$, se tiene que* ínf $\tilde{F}(x) = 0$.

```
lemma F_Real_0_GLB_in_C1 (...)
    : ∀ x : X, x ∈ C1 → IsGLB (F_Real hT C1 C2 x) 0 := by sorry
```

Por último, tenemos:

Lema 4.18. *Consideremos $k : X \to \mathbb{R}$. Recordemos que C_1 es no vacío. Se tiene que $k(C_1) = \{0\}$.*

```
lemma k_in_C1_is_0 (...)
    (hC1_nonempty : C1 ≠ ∅)
    : k hT C1 C2 '' C1 = {0} := by
```

Demostración. Para ver que $k(C_1) \subseteq \{0\}$, puesto que $k(x) := \text{ínf}\, \tilde{F}(x)$, y ínf $\tilde{F}(x) = 0$ para $x \in C_1$ por el resultado anterior, basta utilizar el el ínfimo es único.

```
  ext r
  constructor
  · intro ⟨x, hx, hr⟩ -- hr : k x = r
    rw [← hr]
    apply IsGLB.unique (k_prop hT C1 C2 x)
    exact F_Real_0_GLB_in_C1 hT C1 C2 hC1 hC2 hC1C2 x hx
```

Para ver que $\{0\} \subseteq k(x)$, necesitamos ver que existe un $x \in C_1$ tal que $k(x) = 0$. Como C_1 es no vacío, sea $x \in C_1$ cualquiera. Entonces $k(x) = 0$, procediendo de la misma forma que en el primer contenido.

```
  · intro hr -- hr : r = 0
    rw [hr]
    obtain ⟨x, hx⟩ := nonempty_has_element C1 hC1_nonempty -- hx : x ∈
  C1
    use x
    constructor
    · exact hx
    · have aux := F_Real_0_GLB_in_C1 hT C1 C2 hC1 hC2 hC1C2 x hx
      have aux' := Classical.choose_spec (F_Real_has_inf hT C1 C2 x)
      exact IsGLB.unique aux' aux
```

□

4.5 La demostración

Vamos a demostrar, por tanto, la implicación principal del lema de Urysohn, siguiendo paso a paso la demostración en Lean.

Sea X un espacio normal. Queremos demostrar que para cada par de cerrados disjuntos no vacíos C_1 y C_2 existe una función continua $f : X \to [0, 1]$ de forma que $f(C_1) = \{0\}$ y $f(C_2) = \{1\}$.

Sean por tanto C_1 y C_2 cerrados con esas condiciones.

```
  · -- →
    intro hT C1 C2 C1nempty C2nempty C1closed C2closed C1C2disj
```

Para empezar, demostramos algunas propiedades auxiliares, como que C_2^c es abierto (por ser C_2 cerrado), y que $C_1 \subseteq C_2^c$ (por ser disjuntos), que nos facilitarán aplicar el resto de resultados que hemos ido probando. También utilizaremos la caracterización de espacios normales (3.14) sobre la hipótesis de X normal.

```
    have C2c_open : IsOpen C2ᶜ := by exact IsClosed.isOpen_compl
    have hC1C2 : C1 ⊆ C2ᶜ := by exact Disjoint.subset_compl_left
      (id (Disjoint.symm C1C2disj))
    rw [characterization_of_normal] at hT
```

Ahora, por comodidad vamos a definir las siguientes funciones.

```
    let G := H hT C1 C2
    let g := fun x ↦ k hT C1 C2 x
```

De esta forma no tenemos que arrastrar los argumentos de H y k en cada paso. Cambio el nombre de H a G para seguir con la notación del esquema, puesto que es la función de la forma $G : \mathbb{N} \to \mathcal{P}(X)$. También cambio el nombre de k a g, para evitar confusiones.

Ahora, recordemos que $k : X \to \mathbb{R}$ no era exactamente la función que buscábamos. Vamos a definir finalmente la función a utilizar para separar nuestros cerrados, f, de la siguiente manera:

```
let f : X → Y := fun x ↦ ⟨g x, by
    rw [hY]; exact k_in_01 hT C1 C2 x⟩
```

Es decir, f es una función que toma valores en X y devuelve un valor real y una demostración de que este valor está en realidad en el intervalo $[0, 1]$, dada utilizando `k_in_01`, que habíamos demostrado antes. Por tanto, Lean la interpreta como una función $f : X \to [0, 1]$, que es justo lo que queríamos.

Así que ya tenemos la función separadora y podemos dar el gran paso:

```
use f
```

Nos queda por demostrar que f es continua, que $f(C_1) = \{0\}$ y que $f(C_1) = \{1\}$.

```
constructor
```

4.5.1 Continuidad de f

Queremos ver que f es continua. Para ello, aplicamos el resultado `continuousInSubspace_iff_trueForBase`, que vimos que era combinación de 3.12 y 3.13.

```
· rw [@continuousInSubspace_iff_trueForBase
    X ℝ Y T T' R hR f
    {s | ∃ a b : ℝ, s = Set.Ioo a b}
    (by exact BaseOfRealTopo hT')]
```

Luego basta demostrar que para cada abierto básico W de $\mathbb{R}$, $f^{-1}(W)$ es abierto en X. Sea $W = (a, b)$ un abierto de la base formada por los intervalos abiertos reales.

```
  intro W hW
  obtain ⟨a, b, hW⟩ := hW -- hW : W = (a, b)
```

Queremos ver que $f^{-1}(W)$ es abierto en X. Utilizando la caracterización de abiertos (3.1), basta ver que es entorno de todos sus puntos. Sea entonces $x \in f^{-1}(W)$, lo que quiere decir que $f(x) \in (a, b)$.

```
  rw [A_open_iff_neighbourhood_of_all]
  intro x hx
  rw [Set.mem_preimage, hW] at hx -- h x : f x ∈ (a, b)
```

Puesto que $f(x) \in (a, b)$, se tiene que existe $p \in \mathbb{Q}$ con $a < p < f(x)$, y también existe $q \in \mathbb{Q}$ con $f(x) < q < b$.

```
obtain ⟨p, hp⟩ := exists_rat_btwn hx.left
obtain ⟨q, hq⟩ := exists_rat_btwn hx.right
```

Vamos a demostrar que $x \in G(q)$ y que $x \notin \overline{G(p)}$.

Primero recordemos los resultados 4.13 y 4.14 (con los nombres de las funciones correspondientes a la notación actual):

- `k_claim1`: $\forall p \in \mathbb{Q}, \forall x \in X, x \in \overline{G(p)} \implies g(x) \leq p$
- `k_claim2`: $\forall p \in \mathbb{Q}, \forall x \in X, x \notin G(p) \implies g(x) \geq p$

```
have claim1 := k_claim1 (...)
have claim2 := k_claim2 (...)
```

(1) Para ver que $x \notin \overline{G(p)}$, por reducción al absurdo supongamos que sí está. Entonces podemos aplicar `claim1` y obtener que $g(x) \leq p$ pero teníamos $p < f(x) = g(x)$, contradicción.

```
have aux1 : x ∉ closure (G p)
· by_contra c
  apply claim1 p x at c
  linarith
```

(2) De manera similar, si suponemos por reducción al absurdo que $x \notin U(q)$, entonces por `claim2` obtenemos que $g(x) \geq q$, pero teníamos $q > f(x) = g(x)$, contradicción.

```
have aux2 : x ∈ G q
· by_contra c
  apply claim2 q x at c
  linarith
```

Recapitulando, queremos ver que $f^{-1}(W)$ es entorno de x (x era arbitrario). Esto es, por definición, encontrar un conjunto $U \subseteq X$ que sea abierto y de manera que $x \in U \subseteq f^{-1}(W)$.

Tomemos el conjunto $U = G(q) \cap (\overline{G(p)})^c$ y veamos que cumple estas condiciones.

```
use (G q) ∩ (closure (G p))ᶜ
constructor
```

(1) Veamos que $U \subseteq f^{-1}(W)$. Para ello, sea $y \in U$ y veamos que $y \in f^{-1}(W) = f^{-1}((a, b))$, es decir, veamos que $a < f(y) < b$.

```
    · intro y hy
      rw [hW]
      constructor
```

Se tiene que $y \notin G(p)$, pues en caso contrario, $y \in G(p) \subseteq \overline{G(p)}$, pero $y \in \overline{G(p)}^c$. Aplicando `claim1`, se tiene que $f(y) = g(y) \geq p > a$.

```
      · have hy : y ∉ G p
        · by_contra c
          apply subset_closure at c
          exact hy.right c
        apply claim2 p y at hy
        linarith
```

Por otro lado, como $y \in G(q) \subseteq \overline{G(q)}$, por `claim2` se tiene que $f(y) = g(y) \leq q < b$.

```
      · have hy := hy.left
        apply subset_closure at hy
        specialize claim1 q y hy
        linarith
```

Luego concluimos que $f(y) \in (a, b) = W$. El trabajo duro ya está hecho, ya solo falta ver que $x \in U$ y que U es abierto.

```
    constructor
```

(2) Como habíamos visto, $x \in G(q)$ y $x \notin \overline{G(p)}$. Luego $x \in U$.

```
    · constructor
      · exact aux2
      · exact aux1
```

(3) Probar que U es abierto es sencillo: como es una intersección finita, basta ver que ambos componentes son abiertos. Por 4.5, $G(q)$ es abierto. Además, $\overline{G(p)}$ es cerrado por ser una clausura, luego su complementario es abierto.

```
    · apply IsOpen.inter
      · exact H_isOpen hT C1 C2 C1closed C2c_open hC1C2 q
      · rw [isOpen_compl_iff]
        exact isClosed_closure
```

4.5.2 Imagen de f

Para terminar la demostración, solo nos falta ver que $f(C_1) = \{0\}$ y que $f(C_2) = \{1\}$.

Para empezar, notemos que $f(A) = g(A)$ para cualquier $A \subseteq X$.

```
have aux : ∀ A : Set X, f '' A = g '' A
  · intro A; ext x; simp
```

Luego podemos reducir el objetivo a $g(C_1) = \{0\}$ y que $g(C_2) = \{1\}$.

El resultado se deduce entonces del lema 4.18 y su análogo para C_2.

```
  constructor
  · exact k_in_C1_is_0 hT C1 C2 C1closed C2c_open hC1C2 C1nempty
  · exact k_in_C2_is_1 hT C1 C2 C1closed C2c_open hC1C2 C2nempty
```

Lo que concluye la prueba del lema de Urysohn.

4.6 La prueba de Mathlib

La prueba del lema de Urysohn que he descrito en este capítulo ha sido implementada por mí de manera completamente independiente a la de Mathlib, la cual no leí hasta haber escrito completamente mi demostración. Para cerrar este capítulo me gustaría hacer una breve comparación de ambas.

La implementación que se encuentra en Mathlib está documentada en [17].

Como comentamos al principio del capítulo, la parte más complicada de la demostración ha sido construir la función G, por requerir inducción completa para construir los abiertos e inducción sobre dos variables para probar sus propiedades. En efecto, en la documentación de Mathlib, escriben:

> La mayoría de las fuentes prueban el lema de Urysohn utilizando una familia de conjuntos abiertos indexada por los números racionales diádicos en $[0, 1]$. Hay muchas dificultades técnicas al formalizar esta demostración (por ejemplo, es necesario formalizar la "inducción diádica", y luego probar que la familia resultante de conjuntos abiertos es monótona).

En efecto, es posible construir la sucesión que nosotros hemos construido tomando sólo los diádicos, pero uno se encuentra con problemas parecidos a los que hemos encontrado nosotros. El objetivo de la prueba de Mathlib, es, por tanto, evitar esta complicación. A continuación veremos un esquema de la demostración de Mathlib.

Sea X un espacio normal. Sea $\mathcal{CU}_X$ el conjunto de los pares normales de X, es decir, los pares de la forma (C, U) con U abierto, C cerrado y $C \subseteq U$. Definimos las funciones:

$$\begin{array}{rccc} L & : & \mathcal{CU}_X & \longrightarrow & \mathcal{CU}_X \\ & & (C,U) & \longmapsto & (C,V) \end{array} \qquad y \qquad \begin{array}{rccc} R & : & \mathcal{CU}_X & \longrightarrow & \mathcal{CU}_X \\ & & (C,V) & \longmapsto & (\overline{V},U) \end{array}$$

Donde V es el resultado de aplicar la caracterización de espacios normales (3.14) al par (C,U) correspondiente.

Además, para cada $(C,U) \in \mathcal{CU}_X$ consideremos la sucesión de funciones $\{f_n^{(C,U)} : X \to \mathbb{R}\}_{n\in\mathbb{N}}$ dada por

$$\begin{cases} f_0^{(C,U)}(x) &= \chi_{U^c}(x) \\ \\ f_{n+1}^{(C,U)}(x) &= \dfrac{f_n^{L(C,U)}(x) + f_n^{R(C,U)}(x)}{2}, \quad n = 0, 1, \ldots \end{cases}$$

Se puede probar que esta es una sucesión monótona de funciones de forma que, para cada par (C,U) y cada $n \in \mathbb{N}$, $f_n^{(C,U)}(x) = 0, \forall x \in C$, $f_n^{(C,U)}(x) = 1, \forall x \in D$, y $f_n^{(C,U)}(x) \in [0,1], \forall x \in X$.

En particular, sean C y D los cerrados disjuntos que queremos separar y consideremos el par $(C, D^c) \in \mathcal{CU}_X$. Sea $f : X \to \mathbb{R}$ la función dada por

$$f(x) = \lim_{n\to\infty} f_n^{(C,D^c)}$$

Entonces f es una función continua que separa C y D.

5 Conclusión

A lo largo de este trabajo se han perseguido tres objetivos principales: aprender a utilizar Lean como asistente de demostración, formalizar un resultado matemático de nivel avanzado, el lema de Urysohn, y, como resultado de ambas, adquirir una comprensión profunda del proceso de verificación formal, sus ventajas y sus dificultades. En esta sección se exponen las principales conclusiones que he extraído de esta experiencia.

5.1 Aprendizaje de Lean

El proceso de aprendizaje de Lean ha resultado muy fluido. La disponibilidad de recursos como el curso de Kevin Buzzard (*Formalising Mathematics*), la documentación oficial y otros materiales creados por la comunidad, como el *Natural Number Game*[17], me han facilitado enormemente los primeros pasos.

Sin embargo, conforme avanzaba, fui detectando un segundo nivel de aprendizaje más sutil: aunque mis soluciones a los ejercicios propuestos por Buzzard eran correctas desde el punto de vista formal (aceptadas por Lean), eran considerablemente más largas y enrevesadas que las soluciones propuestas en el curso.

Esto me hizo ver que, además de aprender a construir demostraciones válidas, hay un proceso adicional de aprendizaje centrado en cómo escribir demostraciones eficientes y elegantes en Lean, lo que requiere un entendimiento más profundo de las tácticas y las buenas prácticas del lenguaje (como evitar el abuso de `simp`).

Esta misma situación se ha repetido durante la formalización de los resultados de topología básica de la segunda sección. Muchas de las demostraciones que aparecen en esta memoria fueron inicialmente mucho más extensas, redundantes o confusas. El trabajo de revisión y simplificación ha sido constante; conforme iba aprendiendo nuevas formas de simplificar problemas, volvía sobre mis pasos para mejorar los resultados anteriores.

Aún así, todavía queda margen de mejora. En comparación con las versiones disponibles en Mathlib, mis demostraciones podrían reducirse todavía más. No obstante, considero que este esfuerzo de simplificación tiene un límite razonable en el contexto de los objetivos de este trabajo, pues optimizar al máximo suele ir en detrimento de la legibilidad del código. Algunas demostraciones de Mathlib son muy difíciles de seguir precisamente por su alto nivel de compacidad.

Durante esta etapa también comprendí la importancia de reutilizar las definicio-

[17] El *Natural number game* es un juego interactivo dedicado a aprender a trabajar con números naturales en el modo táctico de Lean. Ver: `https://adam.math.hhu.de/#/g/leanprover-community/nng4`.

nes existentes en Mathlib. Aunque al principio definí por mi cuenta conceptos como la continuidad, la clausura o espacios normales, más adelante pasé a utilizar las definiciones estándar de la biblioteca, que resultaba mucho más práctico.

Este proceso también me obligó a consolidar mi comprensión de los conceptos matemáticos con los que estaba trabajando, ya que formalizarlos exige enfrentarse a sus definiciones con un nivel de precisión y detalle que habitualmente no se requiere al trabajar en papel. Además, obtuve una idea bastante clara de cómo se pueden formalizar ciertos objetos al comparar mis definiciones con las de la librería que utilicé posteriormente.

Por último, esta transición me permitió empezar a desarrollar cierta intuición sobre qué resultados ya están formalizados en Mathlib y de qué manera pueden estar escritos, para poder localizarlos mediante herramientas como `exact?`. También, al encontrarme con las limitaciones de estas herramientas, aprendí a utilizar herramientas más potentes como LeanSearch o Moogle.

Por todo esto, incluir esta parte en el trabajo me parecía especialmente relevante.

5.2 Formalización del lema de Uryshon

La formalización del lema de Urysohn ha sido, sin duda, la parte más exigente del trabajo. Al tratarse de una demostración con ideas matemáticas relativamente complejas, intentar adaptarlas al paradigma de Lean, que es mucho más estructurado y restrictivo, ha supuesto una dificultad añadida. Muchas intuiciones que en papel resultan inmediatas requieren un esfuerzo considerable para ser traducidas al lenguaje formal.

Un ejemplo de esto es la demostración de que la sucesión de abiertos obtenidos era monótona (la propiedad $\star$). En todas las fuentes consultadas, esta propiedad se presenta como algo que se deduce trivialmente de la construcción. Sin embargo, en Lean fue necesario construir esta parte con cuidado, y encontrar la forma adecuada de hacerlo me llevó varios meses.

Por supuesto, una vez completada la formalización, el resultado era un código extenso de cientos de líneas. De la misma forma que en la anterior etapa, lo he ido simplificando, principalmente extrayendo resultados auxiliares fuera del archivo principal para después poder reutilizarlos. A pesar de todo, la prueba de Mathlib que hemos comentado al final sigue siendo significativamente más corta.

Más allá de esto, esta experiencia me ha hecho ver que existe una cierta forma de pensar que se desarrolla con el tiempo al trabajar con Lean: no se trata simplemente de traducir la demostración del papel línea por línea, sino de encontrar una formulación más adecuada computacionalmente. Por ejemplo, la topología en Mathlib se formaliza, en general, haciendo uso del lenguaje de filtros. En mi

caso, al no estar familiarizada con esta herramienta, opté por una estrategia alternativa que se adaptara mejor a mis conocimientos, pero que finalmente ha resultado ser mucho menos eficiente.

En cualquier caso, este esfuerzo ha sido una gran oportunidad para mejorar mi manejo del lenguaje. Una parte especialmente difícil ha sido la utilización de distintos tipos de inducción a lo largo de la prueba. Como muchas de las ideas iniciales que tuve no funcionaban, terminé escribiendo y demostrando distintas variantes de inducción para comprender por qué ciertas aproximaciones no eran válidas y cómo adaptarlas correctamente.

Otro aspecto que me resultó complejo fue colaborar con la inferencia de tipos de Lean. Aunque en general Lean resuelve automáticamente muchos detalles del tipado, ahorrando mucho trabajo, en ocasiones esta automatización se convierte en un obstáculo. Un ejemplo claro de esto ha sido tener que definir una versión modificada de la función F para que devolviera conjuntos en $\mathbb{R}$ en lugar de en $\mathbb{Q}$ y poder así definir su ínfimo real.

Sin embargo, esta dificultad también me ha llevado a aprender a utilizar herramientas especialmente útiles como la táctica `exact_mod_cast` o la función `Subtype.val` de Mathlib.

En resumen, este proceso ha sido clave para consolidar mis conocimientos sobre Lean y entender cómo afrontar la formalización de resultados matemáticos complejos dentro de un entorno asistido.

5.3 Posibles mejoras y ampliaciones

Existen varias direcciones en las que podría ampliarse o mejorar el trabajo realizado:

- **Uso sitemático de definciones de Mathlib**

  Aunque en versiones iniciales utilicé definiciones propias para ciertos conceptos, más adelante las reemplacé por las definiciones de Mathlib. Sin embargo, aún quedan algunos objetos que no he logrado integrar completamente, como la definición de subespacio topológico o la de topología usual.

  El objetivo de este trabajo no era principalmente conseguir que el resultado fuera reutilizable en otros desarrollos, ya que la formalización del lema de Urysohn ya existe en Mathlib, sino entender en detalle el proceso de formalización y reflexionar sobre las dificultades que pueden surgir. No obstante, sería interesante reescribir la demostración de manera que se utilicen exclusivamente definiciones y construcciones estándar de Mathlib, para profundizar en sus implementaciones y modos de uso.

  En particular, se propone expresar el teorema en la forma siguiente:

```
theorem Urysohn {X : Type} [T : TopologicalSpace X]
  [N : NormalSpace X] {s t : Set X} (hs : IsClosed s)
  (ht : IsClosed t) (hd : Disjoint s t) :
  ∃ f : X → ℝ, Continuous f ∧ Set.EqOn f 0 s ∧ Set.EqOn f 1 t
    ∧ ∀ x, f x ∈ Set.Icc 0 1
```

En este enunciado, la topología que acompaña al espacio de los números reales se asume automáticamente en la definición de continuidad mediante una instancia que Lean infiere por defecto. Por tanto, únicamente faltaría comprobar que dicha instancia se corresponde con la definición que he utilizado de topología usual, demostrando este resultado:

```
lemma my_usual_equiv : @UniformSpace.toTopologicalSpace ℝ
  (by exact PseudoEMetricSpace.toUniformSpace) = UsualTopology
```

- **Documentación del código**

  Una posible mejora sería escribir la documentación del código siguiendo el estilo utilizado en la de Mathlib. La comunidad de Lean ofrece una herramienta que genera este tipo de documentación automáticamente a partir de los comentarios escritos en los archivos `.lean`[18]. Otra opción podría ser utilizar una herramienta como *leanblueprint*, que permite crear y publicar de manera automatizada una presentación del proyecto en forma de "blueprint"[19].

- **Topology Game**

  Una línea interesante sería diseñar un recurso similar al *Natural Number Game*, pero centrado en introducir los conceptos básicos de topología formalizados en Lean. Esto no solo reforzaría el aprendizaje propio, sino que podría resultar útil para otras personas interesadas en iniciarse en Lean en el contexto de la topología. De hecho, ya existe un juego con esta idea[20], pero utiliza la versión Lean 3, por lo que sería interesante actualizarlo.

  Además, este tipo de juegos se desarrollan directamente en Lean utilizando una base ya disponible[21], lo que ofrece también la oportunidad de explorar Lean como lenguaje de programación funcional, más allá de su uso como verificador de demostraciones.

- **Lean como verificador de programas**

  Por último, cabe mencionar que he sido admitida en el Máster en Métodos Formales en Ingeniería Informática (UCM con UPM). Esto me permitirá continuar explorando Lean y otros verificadores desde una nueva perspectiva: el uso de estas herramientas en la verificación de programas y

[18] Ver: `https://github.com/leanprover-community/doc-gen`.
[19] Ver: `https://github.com/PatrickMassot/leanblueprint`.
[20] Ver: `https://mmasdeu.github.io/topologygame/`
[21] Ver:
`https://github.com/leanprover-community/lean4game/blob/main/doc/create\_game.md`.

especificaciones formales. Creo que esta formación complementará de forma natural el trabajo realizado en este proyecto, ya que me permitirá profundizar en otra faceta relevante y actual de los sistemas de verificación.

Referencias

[1] Trieu H Trinh, Yuhuai Wu, Quoc V Le, He He, and Thang Luong. *Solving olympiad geometry without human demonstrations. Nature*, 625(7995):476–482, 2024.

[2] Herman Geuvers. *Proof assistants: History, ideas and future. Sadhana*, 34:3–25, 2009.

[3] Imperial College London Kevin Buzzard. *Formalising Mathematics 2024*, 2024. Ver `https://github.com/ImperialCollegeLondon/formalising-mathematics-2024`. Última consulta: 22 de junio de 2025.

[4] Jeremy Avigad, Leonardo De Moura, Soonho Kong, Sebastian Ullrich, and contributors from the Lean Community. *Theorem Proving in Lean 4*, 2024. Ver `https://leanprover.github.io/theorem_proving_in_lean4`. Última consulta: 23 de abril de 2025.

[5] Lean Prover Community. *mathlib4: The Lean4 Mathematical Library*. Ver `https://github.com/leanprover-community/mathlib4`. Última consulta: 23 de abril de 2025.

[6] Christopher A. Bailey and contributors from the Lean Community. *Type Checking in Lean 4*, 2024. Ver `https://ammkrn.github.io/type_checking_in_lean4`. Última consulta: 23 de abril de 2025.

[7] Jeremy Avigad, Leonardo De Moura, and Soonho Kong. *Theorem proving in Lean*. 2021.

[8] Morten Heine Sørensen and Pawel Urzyczyn. *Lectures on the Curry-Howard isomorphism*, volume 149. Elsevier, 2006. Chapter 5. The Untyped Lambda Calculus.

[9] Thierry Coquand and Gérard Huet. *The calculus of constructions*. PhD thesis, INRIA, 1986.

[10] Benjamin C Pierce. *Types and programming languages*. MIT press, 2002.

[11] Mario Carneiro. *The Type Theory of Lean*. 2019. Section 5. Reduction of inductive types to W-types.

[12] Mario Carneiro. *Lean4Lean: Towards a formalized metatheory for the Lean theorem prover. arXiv e-prints*, pages arXiv–2403, 2024.

[13] Lean Prover Community. *Mathlib manual: Tactics.*, 2023. Ver `https://leanprover-community.github.io/mathlib-manual/html-multi/Tactics/#tactics`. Última consulta: 24 de junio de 2025.

[14] Morph. *Moogle, a semantic search engine for Mathlib, the Lean mathematical library*, 2023. Ver `https://www.moogle.ai/`. Última consulta: 24 de mayo de 2025.

[15] Guoxiong Gao, Haocheng Ju, Jiedong Jiang, Zihan Qin, and Bin Dong. *A semantic search engine for Mathlib4. arXiv preprint arXiv:2403.13310*, 2024. Ver `https://leansearch.net/`. Última consulta: 24 de mayo de 2025.

[16] Stephen Willard. *General topology.* Courier Corporation, 2012.

[17] Lean Prover Community. *Urysohn's lemma*, 2023. Ver `https://leanprover-community.github.io/mathlib4_docs/Mathlib/Topology/UrysohnsLemma.html`. Última consulta: 22 de junio de 2025.