Demostración Asistida de Teoremas con Lean

UNIVERSIDAD
COMPLUTENSE
MADRID

Autor
Santiago Mourenza Rivero
Directores
Ángel González Prieto
María Inés de Frutos Fernández
Facultad de Ciencias Matemáticas
Universidad Complutense de Madrid

Trabajo de Fin de Grado presentado para optar al grado en
Matemáticas

Julio 2024

Dedicatoria

A mi amigo Alex por ser una de esas personas en las que siempre se puede contar. Y a mi hermano, Rodri; a mi madre y a mi padre por ser un pilar durante estos 5 años de carrera.

Agradecimientos

Agradezco a mis tutores Ángel y María Inés por haberme ayudado a lo largo de este trabajo durante todo este año. A Miguel Ángel y a Alex que han estado conmigo en llamada durante esas tardes en las que el código no salía. Y junto con ellos a mi familia por leerse todas las páginas de este documento.

Índice general

Resumen

A lo largo de este documento se va a explicar la teoría de tipos simples y la teoría de tipos dependientes con la intención de comprender los conceptos teóricos en los que se fundamenta el lenguaje Lean 4 para formalizar teoremas. También se ahonda en comprender la sintaxis y fundamentos de Lean 4, pudiendo probar así diferentes resultados sencillos. Una vez comprendidos todos los aspectos básicos de Lean 4 con explicaciones y ejemplos pasamos a demostrar la equivalencia entre el Principio de buena ordenación, el Axioma de elección y el Lema de Zorn de manera clásica para luego formalizar algunas de estas pruebas en Lean 4. Estas dos formas de demostrar los mismos resultados nos ayudan a ver la utilidad de Lean 4 frente a las demostraciones en papel así como las dificultades de formalizar con total precisión, que es la forma que sigue Lean 4 para asegurar la corrección de una prueba.

Palabras clave

Lean 4, Mathlib 4, Lema Zorn, Principio Buena Ordenación, Axioma Elección, Tácticas.

Abstract

Throughout this document we will explain the theory of simple types and the theory of dependent types with the intention of understanding the theoretical concepts on which the Lean 4 language is based to formalize theorems. We also delve into understanding the syntax and fundamentals of Lean 4, thus being able to prove different simple results. Once all the basic aspects of Lean 4 are understood with explanations and examples, we go on to demonstrate the equivalence between the Well Ordering Principle, the Axiom of Choice and Zorn's Lemma in a classical way and then formalize some of these proofs in Lean 4. These two ways of demonstrating the same results help us to see the usefulness of Lean 4 versus paper demonstrations as well as the difficulties of formalizing with total precision, which is the way that Lean 4 follows to ensure the correctness of a proof.

Keywords

Lean 4, Mathlib 4, Zorn's Lemma, Well-Ordering Principle, Axiom of Choice, Tactics.

Capítulo 1

Introducción y objetivos

El avance de la tecnología ha impactado significativamente en muchos campos de estudio, y las matemáticas no son una excepción. Uno de los desarrollos más interesantes en esta área ha sido la creación de asistentes de demostración. Estas son herramientas que permiten la formalización y verificación de demostraciones matemáticas, garantizando un alto grado de rigor y precisión. Entre estos asistentes, Lean 4 ha surgido como una de las opciones más prometedoras, junto con otros sistemas como Coq, Isabelle y Agda.

Lean 4 es la última versión del asistente de demostración Lean, desarrollado inicialmente por Leonardo de Moura en Microsoft Research en 2013. Este sistema ha sido diseñado para proporcionar un entorno interactivo (*infoview*) en el que los matemáticos puedan escribir y verificar sus demostraciones de la manera más formal posible. Lean 4 es un lenguaje de programación funcional basado en el cálculo lambda, que permite a los usuarios definir estructuras de datos, funciones y teoremas utilizando una sintaxis similar a la utilizada en matemáticas.

Lean 4 se basa en un núcleo lógico llamado "*Calculus of Inductive Constructions*"(CIC), que es una extensión del cálculo de construcciones con soporte para tipos inductivos. Este núcleo permite la representación precisa de conceptos matemáticos complejos y la verificación rigurosa de demostraciones. Además, Lean 4 incluye características avanzadas como metaprogramación, que permite a los usuarios escribir programas que manipulan otros programas, facilitando la automatización de tareas repetitivas en la formalización de matemáticas.

Uno de los principales logros de Lean 4 es su capacidad para formalizar y verificar teoremas complejos. Por ejemplo, el teorema de los números primos, el teorema de los cuatro colores y partes significativas del teorema de Feit-Thompson han sido formalizados utilizando Lean. Estos resultados muestran la potencia y la utilidad de Lean 4 para manejar demostraciones matemáticas de alto nivel.

Además de garantizar la corrección de los resultados probados, los asistentes de demostración

como Lean 4 ofrecen mayor claridad y rigor en la formulación de teoremas y definiciones. Los matemáticos pueden beneficiarse de un entorno interactivo que les permite explorar diferentes enfoques y recibir retroalimentación sobre sus decisiones en tiempo real. Esto no solo mejora la precisión de las demostraciones, sino que también abre la puerta a la colaboración interdisciplinaria entre matemáticos e informáticos, fomentando un ámbito de desarrollo ideal para profesionales de ambos campos.

1.1. Objetivos

En este trabajo vamos a buscar ejemplificar la manera de proceder para, dado un resultado conocido y demostrado de forma clásica en papel, modelizarlo en Lean 4 y la lógica que se sigue para ello. Por lo tanto se tienen los siguientes objetivos para llegar a esta meta:

- Entender el fundamento teórico de Lean 4 y su funcionamiento mediante resultados sencillos.
- Conocer las tácticas más conocidas de Lean 4 así como sus usos.
- Entender un resultado conocido y su demostración clásica para tener un enfoque de como formalizarlo.
- Formalizar parte del resultado y explicar la lógica subyacente.

1.2. Plan de trabajo

El camino para llegar a la meta que buscamos consta de varias etapas de trabajo que vamos a seguir para mantener una estructura que nos ayude a cumplir los objetivos, principalmente el de formalizar en Lean 4 un resultado conocido. Las etapas a seguir son las siguientes:

- **Etapa 1 - Base teórica de Lean:** Estudiar la estructura de Lean 4 y la teoría en la que se basa para afirmar su corrección.
- **Etapa 2 - Comprensión y uso del lenguaje:** Comprender los conceptos necesarios del lenguaje para abordar diferentes tipos de pruebas.
- **Etapa 3 - Formalización del resultado:** Estudiar la equivalencia del Lema de Zorn, principio de buena ordenación y Axioma de Elección para formalizar en Lean 4 alguna de las implicaciones.

Capítulo 2

Introducción a la teoría de tipos

¿Qué es Lean?

Lean es una herramienta para formalizar expresiones complejas basándose en la teoría de tipos dependientes. En Lean toda expresión va a tener un tipo asociado y en esta sección vamos a ver todo lo referente a la lógica que tiene detrás para ver más adelante el uso de herramientas para hacer demostraciones. A lo largo de este capítulo y de los siguientes se ha utilizado el libro [1] de donde se inspiran algunos de los ejemplos en los que nos apoyaremos para entender la parte teórica.

2.1. Teoría de tipos simple

Una teoría de tipos es una colección de reglas de inferencia que resultan en asertos. El nombre de teoría de tipos viene del hecho de que hay asertos que definen tipos, asignándolos a una colección de objetos formales que vamos a llamar términos. La sintaxis para los términos es **término : tipo**.

Definición 2.1.1 (Término)**.** Un término en lógica se define como un símbolo constante, una variable o una función que dado un término devuelva otro término.

Los tipos más básicos se suelen denominar átomos o tipos atómicos. Algunos de los que se suelen definir de esta manera son los naturales y los booleanos, que al final de esta sección vamos a definir. Por ejemplo, 42 : **Nat** es un término constante e y **: Bool** una variable.

Otro de los tipos más importantes son los tipos funcionales, estos se representan como $T_1 \to T_2$ que se traduce en que dado un elemento $t_1 : T_1$ devuelve otro $t_2 : T_2$. El operador $\to$ asocia por la derecha y un ejemplo de tipo funcional puede ser add : **Nat** $\to$ **Nat** $\to$ **Nat**, formalmente hablando es una función que dado un natural devuelve una función de tipo **Nat** $\to$ **Nat**.

Los tipos funcionales se pueden crear utilizando lambda-abstracciones. Estos términos se construyen de forma inductiva al igual que las funciones. Una lambda abstracción se representa como

$\lambda x.t$, donde $x : T_1$ es una variable, $t : T_2$ es un término y su tipo es $T_1 \to T_2$. Un ejemplo sencillo utilizando el tipo funcional anterior sería $(\lambda x.\ \text{add}\ x\ x) : \mathbf{Nat} \to \mathbf{Nat}$, en donde $x : \mathbf{Nat}$ y add $x\ x$: **Nat**.

Antes de ver la sintaxis de las reglas de inferencia definamos los asertos así como su notación. Los asertos pueden tener hipótesis previas necesarias para que sea cierto. Vamos a utilizar el símbolo $\vdash$ para separar las hipótesis de la conclusión. El contexto es la lista de hipótesis necesarias para que un aserto sea cierto, se denota por Γ.

Definición 2.1.2 (Aserto)**.** Un aserto es una construcción lógica que dado un contexto de hipótesis (Γ) afirma que un hecho es cierto. La forma de escribir un aserto es $\Gamma \vdash$ hecho.

Para poder construir pruebas vamos a necesitar definir el concepto de reglas de inferencia. Una prueba va a ser un conjunto de estas reglas aplicadas en un orden, esto se va a llamar árbol de derivación.

Definición 2.1.3 (Regla de inferencia)**.** Una regla de inferencia se define como la forma lógica que partiendo de unas premisas (asertos conocidos) devuelve una conclusión (nuevos asertos).

Veamos la sintaxis de las reglas de inferencia con un ejemplo sencillo:

Ejemplo 2.1.4.

$$\frac{\Gamma \vdash t : T_1 \qquad \Gamma \vdash T_1 = T_2}{\Gamma \vdash t : T_2}$$

Esta regla nos dice que dado un término t de tipo T_1 y la hipótesis de que el tipo T_1 es igual al tipo T_2 entonces el término t es de tipo T_2.

Para la teoría de tipos que se va a seguir en Lean, es importante que haya ciertos tipos que sean interpretados como proposiciones, es decir, afirmaciones que pueden probarse y que los términos de este tipo sean esas pruebas de dichas proposiciones. Este concepto lo veremos con el tipo **Prop**, que será explicado más adelante. Partiendo de lo que acabamos de decir, es necesario definir tipos que funcionen como conectores para generar así un álgebra booleana. Veamos las reglas de inferencia necesarias para crear estos tipos basándonos en las defininas en el capítulo 1 de [10].

- **Verdad (Tipo unidad)**

  La regla de introducción de la verdad se define como:

$$\frac{}{\Gamma \vdash \text{triv} : \top}$$

  En este contexto, triv es un término que que representa una prueba de $\top$.

- **Falsedad (Tipo vacío)**

  La regla de eliminación de la falsedad se define como:

$$\frac{\Gamma \vdash M : \bot}{\Gamma \vdash \text{absurd}(M) : A}$$

  En este contexto, absurd(M) significa que desde una contradicción se puede derivar cualquier tipo.

- **Implicación (Tipo función)**

  La regla de introducción de la implicación se define tomando como hipótesis que de A puedo derivar B, entonces podemos derivar $A \to B$, la regla es:

$$\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x : A.M : A \to B}$$

  La notación $\lambda x : A.M$ representa una función que toma un argumento x de tipo A y devuelve uno de tipo B, veremos más en detalle esto a continuación y su utilidad en Lean en la sección 2.3 (lambda-abstracciones).

  La regla de eliminación de la implicación se define tomando como hipótesis que se tiene una prueba de que $A \to B$ y una prueba de A, entonces se puede derivar una prueba de B. La regla es:

$$\frac{\Gamma \vdash M : A \to B \quad \Gamma \vdash N : A}{\Gamma \vdash (MN) : B}$$

  La notación (MN) representa la aplicación de la función M al argumento N.

- **Negación (Tipo función a falsedad)**

  La regla de introducción de la negación se define tomando como hipótesis que A es cierto y se deriva una contradicción. La regla es,

$$\frac{\Gamma, x : A \vdash M : \bot}{\Gamma \vdash \lambda x : A.M : \neg A}$$

  La regla de la eliminación de la negación se define tomando que se tiene una prueba de A y otra de $\neg A$ y de esto se deriva una contradicción. La regla es,

$$\frac{\Gamma \vdash M : A \quad \Gamma \vdash N : \neg A}{\Gamma \vdash (NM) : \bot}$$

  Teniendo en cuenta las reglas de la implicación podemos ver que $\neg A$ puede escribirse como $A \to \bot$ que es la manera en la que derivaremos contradicciones en Lean.

- **Conjunción (Tipo producto)**

  La regla de la introducción de la conjunción se define tomando una prueba de A y otra de B derivando en una prueba de $A \wedge B$. La regla es,

  $$\frac{\Gamma \vdash M : A \quad \Gamma \vdash N : B}{\Gamma \vdash \langle M, N \rangle : A \wedge B}$$

  En este contexto, $\langle M, N \rangle$ es un par que representa la conjunción de M y N.

  Las reglas de eliminación de la conjunción se definen tomando una prueba de $A \wedge B$ entonces se puede derivar una prueba de A y, por separado, una prueba de B. Las reglas son las que siguen:

  $$\frac{\Gamma \vdash M : A \wedge B}{\Gamma \vdash \text{fst}(M) : A} \qquad \frac{\Gamma \vdash M : A \wedge B}{\Gamma \vdash \text{snd}(M) : B}$$

  En este contexto, $\text{fst}(M)$ y $\text{snd}(M)$ son respectivamente la primera y segunda componente del par M.

- **Disyunción (Tipo suma)**

  Las reglas de introducción de la disyunción se definen tomando una prueba de A o una prueba de B entonces se deriva una prueba de $A \vee B$. Las reglas son:

  $$\frac{\Gamma \vdash M : A}{\Gamma \vdash \text{inl}(M) : A \vee B} \qquad \frac{\Gamma \vdash M : B}{\Gamma \vdash \text{inr}(M) : A \vee B}$$

  En este contexto, $\text{inl}(M)$ y $\text{inr}(M)$ indican que M es una prueba de A o B respectivamente.

  La regla de eliminación de la disyunción se define tomando una prueba de $A \vee B$, y se puede derivar C tomando como hipótesis A o B entonces se puede derivar C. La regla es:

  $$\frac{\Gamma \vdash M : A \vee B \quad \Gamma, x : A \vdash N : C \quad \Gamma, y : B \vdash P : C}{\Gamma \vdash \text{case } M \text{ of } (\ \text{inl}(x) \implies N |\ \text{inr}(y) \implies P) : C}$$

  La estructura de la conclusión es un análisis de casos sobre M. Si M es una prueba de A entonces usamos la derivación N para probar C, en cambio si M es una prueba de B utilizamos la derivación P para probar C.

Observación 2.1.5. La conjunción y disyunción se les da el nombre de tipo producto y tipo suma por su representación en teoría de conjuntos.

- **Producto cartesiano** El tipo $A \wedge B$ representa un par ordenado. Esto se corresponde con la idea de un producto cartesiano de conjuntos:

  $$A \times B = \{(a, b) | a \in A \text{ y } b \in B\}$$

- **Unión disjunta** El tipo $A \vee B$ representa un tipo que puede ser un valor de tipo A o un valor de tipo B. Esto se corresponde con la unión disjunta de conjuntos:

$$A + B = \{(\text{inl}(a), a) | a \in A\} \cup \{(\text{inl}(b), b) | b \in B\}$$

  En este contexto, inl e inr indican de cual de los conjuntos originales proviene cada elemento.

Además de los tipos que acabamos de ver hay otros dos muy comunes como los booleanos y los naturales, que se definen como siguen:

- **Booleanos**

  Los booleanos tienen dos términos, **true** y **false** y se escribe como **Bool**. Es importante no confundir con las proposiciones, representadas por $\top$ y $\bot$, que en Lean son **True** y **False** respectivamente, de ahí la posible confusión.

- **Naturales**

  Los naturales, nombrados **Nat**, vamos a implementarlos utilizando el término 0 como base y de forma inductiva mediante la función sucesor S : **Nat** $\to$ **Nat** se representan el resto de ellos.

2.2. Tipos como objetos

Acabamos de ver la teoría que subyace a la implementación en Lean 4 en lo que respecta a tipos simples. A continuación vamos a ver de que manera se tratan los tipos en el caso particular de este lenguaje y de que forma se definen los booleanos y naturales. Esta sección está extraída del capítulo 2 de [2] (*Types as objects*) y de la documentación de la librería Matlib4.

Dejando por ahora los tipos que sirven como proposiciones que veremos en el capítulo 5. En primer lugar vamos a decir que **Nat : Sort 1** y **Bool : Sort 1**, esto significa que los naturales y booleanos son objetos de tipo **Sort 1**. A partir de aquí vamos a decir que **Sort 1 = Type 0** o solamente **Type** para abreviar. Esto es únicamente azúcar sintáctico para el universo de tipos **Sort n** si $n \geq 1$ reservando el **Sort 0** o **Sort** para el tipo proposicional **Prop**.

Para entenderlo de mejor manera vamos a asumir que los tipos no son más que conjuntos a nivel teórico. Por lo tanto los elementos de un tipo en particular son los elementos de ese conjunto. Como acabamos de decir tenemos una jerarquía infinita de tipos de manera que **Type**[1] (equivalentemente **Type 0**) es un universo pequeño de tipos. **Type 1** es un universo mayor que contiene a **Type 0** y así sucesivamente con el resto de los números naturales. Un ejemplo del uso de los tipos es el

[1] El tipo de **Type n** es **Type (n + 1)**

producto, sean α : **Type 1** y β : **Type 2** entonces el producto resultante será $\alpha \times \beta$: **Type 2** que es el resultado del **Type (max 1 2)**. En el caso de no querer definir un tipo en específico se nos permite la opción de escribir tipos arbitrarios **Type** _ o **Type***.

Dentro de Lean 4 existen unos tipos especiales que se llaman tipos inductivos. Estos se construyen a partir de una lista de constructores. Vamos a explicar como definimos **Bool**, **Nat**, el tipo producto y el tipo suma, pero estas construcciones permiten crear infinidad de tipos nuevos.

El tipo inductivo **Bool** es una lista enumerada que toma dos posibles valores como son **true** y **false** que sirve como base para definir las reglas de eliminación e introducción previamente explicadas. La sintaxis es la siguiente:

```
inductive Bool : Type where
  | false : Bool
  | true : Bool
```

El tipo inductivo **Nat** se define como hemos indicado anteriormente, con dos constructores, la diferencia principal con **Bool** es que uno de los constructores necesita un elemento del propio tipo para definirse. El básico es el cero y el resto se obtienen con la función sucesor. En Lean 4 se implementa como sigue:

```
inductive Nat where
  | zero : Nat
  | succ (n : Nat) : Nat
```

Los tipos producto y suma necesitan argumentos para definirse, para el tipo producto la sintaxis va a ser un poco diferente pues se va a pensar como una estructura. Esto es así pues el tipo producto es una tupla de la que queremos obtener sus proyecciones. Esta idea es la misma que utiliza Lean para crear estructuras algebraicas, no obstante aquí estamos hablando de su uso en tipos inductivos. Veamos la implementación de estos dos tipos:

```
structure Prod (α : Type u) (β : Type v) where
    fst : α
    snd : β
```

Esta construcción se resume en que dado un tipo producto $a : \alpha \times \beta$, entonces puedo obtener sus proyecciones utilizando a.fst o b.snd. Posteriormente también utilizaremos la notación a.1 y a.2 que es equivalente a esta.

```
inductive Sum (α : Type u) (β : Type v) where
  | inl (val : α) : Sum α β
  | inr (val : β) : Sum α β
```

Esta construcción sigue con la idea que explicamos anteriormente del tipo suma. Dada la suma disjunta $\alpha \oplus \beta$ un elemento de la misma es bien de la forma .inl a donde $a : \alpha$ o bien es .inj b donde $b : \beta$.

2.3. Lambda-abstracciones y evaluación

Lean proporciona una palabra clave **fun** (puede utilizarse λ también) para crear una función de modo que dada una variable $x : \alpha$ y una expresión $t : \beta$ entonces **fun** $x : \alpha \implies t$ que tiene tipo $\alpha \to \beta$. La información de esta sección se basa en el capítulo 2 de [2] (*Function Abstraction and Evaluation*) y de la documentación de la librería Matlib4. En el siguiente ejemplo se ve las dos posibles representaciones de la función $f(x) = x^2$ utilizando lambda-abstracciones:

```
#check fun x : Nat => x^2
#check λ x : Nat => x^2
```

En Lean utilizamos la expresión **#check** para observar el tipo de la expresión, en estos casos la salida sería **Nat** $\to$ **Nat** pero que en Lean se verá como $\mathbb{N} \to \mathbb{N}$. Esto lo veremos más adelante, en el capítulo 3, cuando expliquemos la *infoview* de Lean para ver las salidas de las diferentes tácticas de Lean.

A las abstracciones lambda las vamos a llamar funciones por conveniencia y por lo tanto vamos a utilizar las expresiones que utilizan la palabra reservada **fun**. Realmente todas las funciones son expresiones lambda en Lean, no obstante, decidimos cambiar la notación para acercarnos más al campo de las matemáticas que al de la computación.

Hasta ahora hemos visto que las funciones tienen una variable única y una expresión asociada, por lo tanto en caso de querer usar más de una variable habría que ir anidando funciones para conseguir el objetivo deseado. Expliquemos el siguiente ejemplo:

```
--Formas de anidar una funcion
#check fun (f : α → β → α ) (x : α) (y : β) => f x y --Corta
#check fun (f : α → β → α ) => fun (x : α) => fun (y : β) => f x y --Larga

--Composicion de funciones
#check fun (α β γ : Type _) (g : β → γ) (f : α → β) (x : α) => g (f x)
```

Las primeras dos líneas del ejemplo anterior representan tres lambdas anidadas que tiene como objetivo aplicar una función a dos argumentos. Pero la idea principal del ejemplo es mostrar una forma compacta de una función lambda con argumentos anidados. A partir de ahora siempre se va a utilizar la forma compacta pero la versión larga nos da más pistas del tipo final de la lambda, que en el ejemplo es $(\alpha \to \beta \to \alpha) \to \alpha \to \beta \to \alpha$. En este ejemplo α : **Type u_1** y β : **Type u_2**, no esta especificado porque lo haremos más adelante cuando definamos las variables.

Ahora en el último ejemplo se han reunido todos los conceptos vistos hasta el momento. Lo primero que hacemos es definir tres tipos arbitrarios que en Lean serán respectivamente **(α : Type u_1) (β : Type u_2) (γ : Type u_3)**, justo después definiremos dos funciones y la variable que aplicaremos. Como resultado obtenemos la composición de f con g aplicada a x. Esta expresión completa tiene el siguiente tipo:

```
(α : Type u_1) → (β : Type u_2) → (γ : Type u_3) → (β → γ) → (α → β) → α → γ
```

Una vez hemos definido estas funciones nos interesa poder aplicarlas a valores determinados. Es por ello por lo que vamos a tomar ejemplos particulares para aplicar la función composición que acabamos de crear. Vamos a utilizar como g la función x^2 y como f la función $x + 1$ y vamos a aplicarla a $4 : \mathbb{N}$. Para este tipo de cosas necesitamos introducir un nuevo comando, **#eval**. Que resulta en la evaluación de una expresión para un valor determinado.

```
#eval (fun (g : ℕ → ℕ) (f : ℕ  → ℕ) (x : ℕ) => g (f x)) (fun x => x^2)
(fun x => x + 1) 4 --25
```

Hemos modificado convenientemente la función composición para que los tipos sean naturales y la evaluación unifique correctamente. Una vez se ha definido la función composición se pasan como argumento la función g y f en formato de abstracción lambda y por último el natural en el que va a ser evaluada.

2.4. Definiciones

En esta sección vamos a comentar los diferentes tipos de definiciones que se pueden hacer en Lean como son las globales y locales. La información que vamos a comentar se basa en el capítulo 2 de [2] (*Definitions*) y de la documentación de la librería Matlib4. Las primeras se utilizan en contextos globales y las segundas se utilizan en ámbitos específicos. Ahora veremos en profundidad el significado de las mismas.

2.4.1. Definiciones globales

Antes de pasar a ver el funcionamiento de las definiciones globales es necesario definir qué son en el contexto de Lean:

Definición 2.4.1. (Definición global) Una definición global es un objeto de Lean que se utiliza para introducir un nuevo identificador que está asociado a una expresión o valor. Esto permite asignar un nombre a una función, valor, tipo, entre otros, para luego referenciarlo y utilizarlo en otros lugares del código.

Para crear definiciones en Lean necesitaremos únicamente la palabra reservada **def** seguido del objeto que queramos definir. La forma exacta es:

def nombre : tipo := cuerpo

Es importante destacar que la gran mayoría de las veces el tipo se infiere directamente del cuerpo pero en ocasiones resulta útil para el usuario tener claro el tipo esperado y en caso de error de tipado poder solucionarlo. Vamos a ver un ejemplo de uso de definiciones:

```
def comp (g : ℕ → ℕ) (f : ℕ → ℕ ) (x : ℕ) := g (f x)
def square (x : ℕ) := x ^ 2
def add_one (x : ℕ):= x + 1
#eval comp square add_one 4 --25
```

Estamos replicando lo que hicimos en la sección anterior con funciones lambda pero utilizando definiciones. La novedad del uso de las mismas es el hecho de poder utilizar las funciones definidas en otras partes del código sin tener que repetir la expresión de la propia función. Al final del ejemplo vemos que utilizando las definiciones podemos evaluar sin escribir explícitamente las expresiones dado que han sido definidas previamente. Resulta interesante repetir lo que acabamos de hacer combinando expresiones lambda con definiciones para ver que se utilizan indistintamente dependiendo de las intenciones del usuario:

```
def comp (g : ℕ → ℕ) (f : ℕ → ℕ ) (x : ℕ) := g (f x)
#eval comp (fun x => x^2) (fun x => x + 1) 4--25
```

2.4.2. Definiciones locales

Otro objeto muy importante son las definiciones locales, y del mismo modo que las globales vamos a definirlas para luego mostrar su utilidad.

Definición 2.4.2 (Definición Local)**.** Es un objeto de Lean con la misma utilidad que una definición global pero en un contexto más limitado.

Las definiciones locales permiten crear nombres temporales que se usan dentro en ámbitos específicos como por ejemplo en otras definiciones. A diferencia de las definiciones que acabamos de ver estas tienen una sintaxis distinta sirviéndose de la palabra reservada **let**. De esta manera podemos expresar estas construcciones mediante la estructura:

let nombre : tipo := cuerpo

Por ejemplo si queremos hacer una definición que no use la composición en general, sino que sea una particularización para las funciones **square** y **add_one**, utilizamos definiciones locales en vez de definir estas funciones en el ámbito global.

Es importante destacar que aunque la funcionalidad a nivel de evaluación es la misma, es decir, sumo uno y luego efectúo el cuadrado, tiene diferencias. Al hacer las tres definiciones podemos utilizarlas en otros teoremas o definiciones para poder efectuar una composición, cuadrado o sumar uno fuera del contexto que aquí estamos desarrollando. En cambio, con definiciones locales estamos consiguiendo la misma funcionalidad pero no permite la misma flexibilidad. Es por esto que depende de lo que queramos probar tendremos que utilizar una opción u otra.

```
def comp_square_add_one (x : ℕ) : ℕ :=
  let square (x : ℕ) : ℕ := x ^ 2
  let add_one (x : ℕ) : ℕ := x + 1
  square (add_one x)
#eval comp_square_add_one 4 --25
```

2.5. Variables explícitas, implícitas, secciones y espacios de nombres

En este apartado vamos a ver la manera de declarar variables y el ámbito, que llamaremos sección, en el que estas son utilizables y los espacios de nombres. La información de esta sección se basa en el capítulo 2 de [2] (*Variables and Sections*) (*Namespaces*) y de la documentación de la librería Matlib4. De esta manera vamos a poder simplificar la formalización de definiciones y teoremas al poder reutilizar variables del mismo tipo sin tener que reescribirlas cada vez que sea necesario utilizarlas.

Definición 2.5.1 (Variable)**.** Es un objeto de Lean que asocia un tipo a un nombre en un contexto. Las variables pueden ser explícitas o implícitas dependiendo de si se quiere especificar al ser utilizadas en funciones o si se quiere que Lean intente inferirlas automáticamente del contexto.

2.5.1. Variables explícitas

Lean nos permite el uso de variables para poder hacer las declaraciones de tipos de manera mucho más compacta. En un primer momento, con ejemplos simples, parece que no se ahorra trabajo pero más adelante cuando formalicemos un resultado más extenso, donde sean necesarias varias hipótesis, observaremos la comodidad de poder agrupar las variables al principio del ámbito para aumentar así la legibilidad. Para poder hacer esto Lean consta de la palabra reservada **variable**. Vamos a volver a definir la **comp**, **add_one** y **square** pero utilizando variables para que de esta manera se observe la funcionalidad de las mismas.

```
variable (f g : ℕ → ℕ)
variable (x : ℕ)

def comp := g (f x)
def square := x ^ 2
def add_one := x + 1
```

Todo lo visto hasta ahora ha sido particularizando todo al tipo $\mathbb{N}$ pero el uso de variables nos permite generalizar todo al uso de tipos que caracterizan Lean, es decir, podemos crear variables de un tipo arbitrario y utilizarlo en todo el ámbito.

Definición 2.5.2 (Sección)**.** Una sección o ámbito de variables es un entorno donde las variables son utilizables. Fuera de la sección a efectos prácticos la variable no existe. El inicio y fin de una sección se indican mediante el uso de las palabras reservadas **section**/**end** respectivamente.

Es importante decir que en caso de no encontrarse dentro de ninguna sección las variables serán siempre visibles por todas las definiciones. Esto se puede entender como una sección global en la que se definen subsecciones para construir objetos los cuales usen variables locales no necesarias en el entorno global. Continuemos con un ejemplo para ver todo esto:

```
section uno

  variable (α β γ : Type _)
  variable (g : β → γ )(f : α → β)
  variable (x : α)

  def comp := g (f x)

end uno

section dos

  #check comp    --Correcto
  #check g (f x) --Error por g, f y x no definido

end dos
```

En el ejemplo vemos que hemos creado dos secciones. En la primera definimos las variables necesarias para definir la composición de funciones para variables de tipos arbitrarios. En la segunda comprobamos que la definición de composición nos da el tipo correcto pero al intentar hacer la composición usando las variables da error por estar fuera de la sección en la que fueron definidas.

2.5.2. Variables implícitas

Hay ocasiones dónde la información se pueden deducir del contexto gracias a las hipótesis o información extra dentro de las definiciones. En este caso se utilizan los argumentos implícitos que en su momento los pondremos en forma de variables implícitas, estos argumentos se escriben mediante un guión bajo (_) que nos indica que el sistema tiene que inferir el tipo del contexto, una forma común de expresar los argumentos en teoría de tipos dependientes, no obstante, hay una manera de no tener que indicar estos argumentos con un guión bajo y es utilizando llaves en las definiciones de tipos y variables en vez de los paréntesis que hemos estado utilizando hasta el momento. Es una sintaxis que permite a Lean comprender que puede inferir del contexto toda la información necesaria reduciendo así el número de condiciones tener en cuenta a la hora de escribir nuevas definiciones. Un problema de los argumentos implícitos es que en ocasiones se pierde información del contexto al ser inferida de manera automática.

```
variable {α β γ : Type u}

def comp (g : β → γ) (f : α → β) (x : α) := g (f x)

#check comp (fun x => x + 1) (fun x => x * x) 1   -- ℕ
#check comp (fun x => x + 1) (fun x => x * x) 1.0 -- Float
```

De esta manera podemos utilizar **comp** sin tener que escribir de forma explícita, por ejemplo, que el tipo es natural. Como el tipo de x en el primer caso es natural por serlo el 1 entonces se infiere que el tipo de la expresión global es $\mathbb{N}$, en cambio el segundo al ser 1.0 de tipo **Float**[2], el tipo inferido por la expresión global es **Float**.

2.5.3. Espacios de nombres

Hemos visto el concepto de sección para las variables. Vamos a ver ahora otra forma de organizar nombres en Lean como son los espacios de nombres:

Definición 2.5.3 (Espacio de nombres)**.** Un espacio de nombres es un entorno donde se pueden agrupar definiciones, declaraciones y elementos de código necesarios con la intención de evitar colisiones de nombres y mejorar la legibilidad. La sintaxis de estos objetos es **namespace** nombre/ **end** nombre.

[2]El tipo float está definido en Lean siguiendo el tipo nativo de punto flotante, correspondiente al éstandar IEEE 754 (binary64) utilizado en el lenguaje C bajo el nombre Double.

Los espacios de nombres tienen una similitud con las secciones en cuanto al tratamiento de variables. A su vez tienen una gran diferencia y es el hecho de que el único espacio de nombres anónimo es el raíz, el resto necesitan un nombre.

Que tengan un nombre no es un capricho de implementación, de hecho resulta útil para poder acceder a los objetos del espacio de nombres. Hay dos formas de acceso a estos objetos, acceso por nombre largo y acceso por nombre corto.

- **Nombre largo**: Sea un espacio de nombres llamado **mi_espacio**, queremos acceder a un objeto llamado dentro de él como **mi_objeto**. Para hacer esto con nombre largo usaremos la notación **mi_espacio.mi_objeto**.
- **Nombre corto**: Sea un espacio de nombres llamado **mi_espacio**, utilizamos el comando **open mi_espacio**. Ahora si queremos acceder al objeto **mi_objeto** únicamente necesitaremos escribir este identificador sin referenciar el espacio de nombres.

Si, por ejemplo, creamos dos espacios de nombres sencillos donde se defina la suma de dos naturales y la suma de dos números en punto flotante bajo el alias **add** podemos utilizar esta suma sin colisiones gracias a la notación de nombre largo. Ahora si sabemos que el contexto de uso va a ser únicamente natural se puede utilizar **open Natural** para no preocuparse más por el espacio de nombres y acceder a la suma únicamente utilizando **add**. Justo debajo tenemos un ejemplo de lo que acabamos de explicar:

```
namespace Natural
  variable (x : ℕ)(y : ℕ)
  def add := x + y
end Natural

namespace Flotante
  variable (x : Float)(y : Float)
  def add := x + y
end Flotante

#eval Natural.add 1 2   --3
#eval Flotante.add 1 2  --3.0

open Natural

#eval add 1 2           --3
```

2.6. Teoría de tipos dependientes

La teoría de tipos dependientes extiende la teoría de tipos simples permitiendo que los tipos puedan depender de los parámetros. Una primera aproximación fácil de entender es decir que un

tipo dependiente es un tipo de una familia indexada de conjuntos. Para aproximarse desde un punto de vista formal vamos a dar una definición de tipo dependiente:

Definición 2.6.1. (**Tipo dependiente**): Sea u un universo de tipos, $(\alpha : u)$ un tipo de este universo. Entonces llamamos $\beta : \alpha \to u$ al tipo tal que para todo $(a : \alpha)$ le asigna un tipo $(\beta(a) : u)$.

Con esta definición se sigue la idea intuitiva de que el tipo $\beta(a)$ varía de forma dependiendo del tipo a. Ahora vamos a ver dos tipos dependientes que son comúnmente utilizados y explicaremos matemáticamente la utilidad que tienen.

2.6.1. Tipo producto dependiente

Una función cuyo valor de retorno varía dependiendo del argumento es una función dependiente y su tipo lo vamos a denominar como tipo producto dependiente.

Definición 2.6.2. (Tipo producto dependiente): Sea α un tipo arbitrario, u un universo de tipos y $\beta : \alpha \to u$ una familia de tipos, construimos el tipo cuyos elementos son funciones cuya imagen depende del argumento de entrada,

$$(x : \alpha) \to \beta\ x,$$

denotaremos a este tipo como $\prod_{x:\alpha} \beta(x)$.

Observación 2.6.3. Cabe destacar que el tipo función que hemos estado utilizando hasta ahora es un caso particular del tipo producto dependiente. Esto se explica atendiendo a que si β es constante entonces $\prod_{x:\alpha} \beta$ es equivalente al tipo $\alpha \to \beta$.

Una vez introducido este tipo es importante detallar que valor nos ofrece a la hora de hacer pruebas. El tipo producto dependiente es lógicamente equivalente a los cuantificadores universales, esto nos permite ver la siguiente equivalencia:

$$\prod_{x:\alpha} \beta(x) \equiv \forall x \in \alpha,\ \beta(x)$$

En Lean 4, el uso de este tipo está asociado con el cuantificador universal. De hecho la teoría que subyace a la definición de las expresiones del tipo $\forall x, P(x)$ se basan en reglas como las que definimos en la teoría de tipos simple. Del mismo modo que ya hicimos vamos a ver de que forma se crea la lógica del tipo producto dependiente, así como su comparativa con la regla del cuantificador universal en lógica de primer orden.

Las reglas del producto dependiente, citadas en [3], son:

- Regla de introducción:

$$\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x : A.M : \Pi x : A.B}$$

  En un contexto Γ donde la variable x es de tipo A y M es una expresión de tipo B en el contexto extendido en el que x es de tipo A se deriva en una función que toma un x de tipo A y devuelve una expresión M, esta función es del tipo producto dependiente.

- Regla de la eliminación:

$$\frac{\Gamma \vdash M : \Pi x : A.B \quad \Gamma \vdash N : A}{\Gamma \vdash MN : B[x := N]}$$

  En un contexto Γ donde M es del tipo producto dependiente y N es de tipo A esto deriva en que M aplicado a N es del tipo resultante de sustituir x por N en B.

La equivalencia se observa al tener las respectivas reglas de introducción y eliminación de el cuantificador universal:

$$\text{Regla de introducción: } \frac{\Gamma \vdash P(a)}{\Gamma \vdash \forall x, P(x)} \tag{2.1}$$

En 2.1 a es un elemento arbitrario cuya única suposición es que pertenece al dominio.

$$\text{Regla de eliminación: } \frac{\Gamma \vdash \forall x, P(x)}{\Gamma \vdash P(a)}$$

Ejemplo 2.6.4. Los vectores con coeficientes reales de tamaño n, $\text{Vec}(\mathbb{R}, n)$ pueden ser expresados como un tipo producto dónde para cada valor $n \in \mathbb{N}$ devuelve un $\mathbb{R}$-Vector de tamaño n y se expresa de la siguiente manera:

$$\prod_{x:\mathbb{N}} \text{Vec}(\mathbb{R}, n)$$

La idea intuitiva para que dentro de Lean tenga utilidad es ver que $\text{Vec}(\mathbb{R}, n)$ es una proposición que determina el hecho "Es un $\mathbb{R}$-vector de tamaño n". Por lo tanto este tipo representa lo que en lógica de primer orden expresaríamos como, para todo n natural podemos construir un vector real de tamaño n, o lo que es equivalente $\forall n \in \mathbb{N},\ \text{Vec}(\mathbb{R}, n)$.

2.6.2. Tipo suma dependiente

Un par dónde el valor de la primera componente sirve como argumento para determinar el valor de la segunda resulta en el tipo dual al previamente descrito y su tipo lo vamos a denominar como tipo suma dependiente.

Definición 2.6.5. (Tipo suma dependiente): Sea α un tipo arbitrario u un universo de tipos y $\beta : \alpha \to \beta$ una familia de tipos, construimos el tipo cuyos elementos son pares cuya segunda componente depende de la primera,

$$(a : \alpha) \times \beta\ a,$$

denotaremos a este tipo como $\sum_{x:\alpha} \beta(x)$.

Observación 2.6.6. Del mismo modo que en el producto, el tipo suma dependiente es una generalización de un tipo que ya hemos utilizado previamente. Entonces en este caso cuando β es una función constante es claro ver que tenemos el tipo $\alpha \times \beta$.

En este tipo dual también nos conviene ver el valor que tiene en las pruebas de Lean. En este caso la suma dependiente es lógicamente equivalente a los cuantificadores existenciales, lo que nos arroja la siguiente equivalencia:

$$\sum_{x:\alpha} \beta(x) \equiv \exists x \in \alpha,\ \beta(x)$$

En Lean 4, el uso de este tipo está asociado con el cuantificador existencial. Del mismo modo que en el caso anterior la expresiones del tipo $\exists x, P(x)$ se basan en reglas. Comparemos ahora las reglas del tipo suma dependiente con las reglas asociadas al cuantificador existencial en lógica de primer orden.

Las reglas de la suma dependiente, citadas en [3], son:

- Regla de introducción:

$$\frac{\Gamma \vdash a : A \qquad \Gamma \vdash b : B[x := a]}{\Gamma \vdash (a, b) : \Sigma x : A.B}$$

  En un contexto Γ donde a es de tipo A y b es de tipo B en un contexto donde $x := a$ entonces esto deriva a la suma dependiente que son pares en los que el tipo del segundo elemento depende del primero.

- Regla de eliminación:

$$\frac{\Gamma \vdash p : \Sigma x : A.B}{\Gamma \vdash \mathrm{fst}(p) : A \qquad \Gamma \vdash \mathrm{snd}(p) : B[x := \mathrm{fst}(p)]}$$

  En un contexto Γ tenemos un elemento p del tipo producto dependiente. Esto deriva en un elemento $\mathrm{fst}(p)$ del tipo A y otro elemento $\mathrm{snd}(p)$ de tipo B donde $x := \mathrm{fst}(p)$.

La equivalencia podemos verla con las respectivas reglas de introducción y eliminación del cuantificador existencial:

$$\text{Regla de introducción: } \frac{\Gamma \vdash P(a)}{\Gamma \vdash \exists x, P(x)}$$

$$\text{Regla de eliminación: } \frac{\Gamma \vdash \exists x, P(x)}{\Gamma \vdash P(a)} \tag{2.2}$$

En 2.2 a no es un elemento arbitrario sino un elemento particular en el cual $P(a)$ es cierto.

Ejemplo 2.6.7. Los números pares, $\text{Par}(n)$ donde n es un número natural, se pueden expresar como un tipo suma dependiente donde la primera componente es un $n \in \mathbb{N}$ y la segunda depende directamente del valor de n, se escribe de la siguiente manera:

$$\sum_{n:\mathbb{N}} \text{Par}(n)$$

La manera de entender esto dentro de Lean es asumir que $\text{Par}(n)$ es una proposición que determina que "El natural n es par". Por lo tanto en lógica de primer orden podemos expresar esto como, existe un natural n tal que dicho natural es par, es decir, $\exists n \in \mathbb{N},\ \text{Par}(n)$.

2.6.3. Correspondencia de Curry-Howard

Acabamos de decir en la sección anterior que el cuantificador universal y existencial son lógicamente equivalentes al tipo producto y suma dependientes. Pero realmente también ocurre esto con los tipos que habíamos definido previamente para las demostraciones de proposiciones en teoría de tipos simple, la correspondencia sería:

Operador Lógico	**Operador Computacional**
Implicación $(A \to B)$	Tipo Función $(A \to B)$
Conjunción $(A \wedge B)$	Producto cartesiano $(A \times B)$
Disyunción $(A \vee B)$	Tipo suma $(A + B)$
Falsedad $(\bot)$	Tipo vacío (False)
Verdad $(\top)$	Tipo unitario (True)
Negación $(\neg A)$	Función a tipo vacío $(A \to \text{False})$

Que esta correspondencia sea cierta es lo que permite que Lean funcione, este resultado es conocido y se conoce como la correspondecia de Curry-Howard o la biyección de Curry-Howard [8](p. 53). El resultado asegura que una prueba hecha con lógica de primer orden puede ser escrita con la teoría de tipos dependientes. Por lo que gracias a esto sabemos que todo lo que hemos dicho hasta el momento, no solo es correcto, sino que permite que las pruebas que vamos a hacer sean formalmente correctas.

Capítulo 3

Básicos de Lean 4

A la hora de hacer pruebas en Lean, al igual que se hace de forma clásica, tenemos que escribir en primer lugar un enunciado y tras ello la prueba. Para esto se utilizará la teoría de tipos de manera que la formalización del enunciado represente de manera fiable lo que se pretende demostrar. La gran diferencia se encuentra en la prueba donde utilizaremos tácticas para ir llegando a la meta que se pretende demostrar. Toda la información relevante para llegar a la meta la veremos en el *infoview* de Lean.

3.1. Pruebas en Lean 4 : infoview y formalización de enunciados

El *infoview* de Lean es una ventana dónde se tendrá toda la información necesaria para saber el estado de la prueba en todo momento. Como las pruebas consisten en un conjunto de tácticas dispuestas en orden, podemos ver en cada paso de la demostración los cambios que se realizan en la meta para así saber si la táctica aplicada avanza en la dirección que se pretende.

La estructura de la ventana va a ser siempre la misma, en primer lugar tenemos el número de metas que tenemos que demostrar, luego las hipótesis y por último la meta asociada a ellas. En el caso de tener más de una vamos a tener una lista de hipótesis con su respectiva meta y se considera como meta actual a la asociada al primer bloque. Veamos para entender de manera clara como sería el estado del *infoview* teniendo abiertas n metas con un conjunto de hipótesis cada una:

```
n goals
case 1
h1 : hip1_1
h2 : hip2_1
...
hm_1: hipm_1_1
⊢ goal 1
...
case n
h1 : hip1_n
h2 : hip2_n
...
hm_n : hipm_n_n
⊢ goal n
```

Una vez conocido esto podemos pasar a formalizar resultados. Es importante definir la estructura de cara a comprender que irá ocurriendo en el *infoview* en cada paso que vayamos dando. Vamos a ir de lo más general a lo más particular y para ello empecemos explicando de que manera enunciaremos resultados y luego la prueba correspondiente.

3.1.1. Enunciados: Introducción al tipo Prop

Los enunciados comienzan con las palabras reservadas **theorem** o **lemma**. Tanto **theorem** como **lemma** hacen la misma función y se utilizan para determinar un resultado con nombre, se le asignará uno u otro dependiendo del contexto ya que realmente no hay ninguna diferencia a la hora de formalizar los resultados. Ambas tendrán un nombre asociado para poder ser utilizadas en otros ámbitos junto con las premisas de lo que se pretende demostrar. Existe otra palabra reservada que no guarda el nombre de la proposición dentro del contexto, esta es **example**, pero internamente es un teorema o un lema. La forma general de un teorema será entonces:

```
theorem nombre_teorema (h1 : hip1) ... (hn : hipn) : enunciado := by
    demostracion
```

Una vez conocida la sintaxis para las proposiciones vamos a entender de que manera funcionan de acuerdo con la teoría de tipos descrita en el Capítulo 2. Introducimos para ello un nuevo tipo, del que ya hemos hablado, al que denominaremos **Prop** que va a representar a todas las proposiciones, como se explica en el capítulo 3 de [2] (*Propositions as Types*). Atendiendo a lo visto anteriormente podemos observar que **Prop** es únicamente azúcar sintáctico para el tipo base que hemos definido como **Sort 0** al igual que los tipos **Type n** son azúcar sintáctico de los definidos como **Sort (n + 1)**. Siendo cierto que el tipo **Prop** va a tener ciertas características especiales, va a mantenerse cerrado bajo en constructor de implicación "→" al igual que los otros universos descritos. De hecho un teorema cualquiera construido como acabamos de ver tiene tipo hip1 → ⋯ → hipn → enunciado.

3.1.2. Demostraciones: Enfoques y correspondencia con Prop

Ahora sabiendo como funcionan los enunciados o proposiciones en Lean, tenemos que entender que son las demostraciones dentro de la teoría de tipos dependientes. Vamos a hacer una definición sencilla en primer lugar, para luego ir dando matices y poniendo dos enfoques diferentes de comprensión de las pruebas en este contexto.

Definición 3.1.1. Sea una proposición $p : Prop$ decimos que f es una prueba de p si es de este tipo, es decir, $f : p$.

Esta definición esta hecha desde un punto de vista constructivo de la lógica y las matemáticas. Básicamente se asume que una proposición es un tipo de datos y una prueba de ella es un objeto correcto de este tipo.

Otra forma de verlo desde el punto de vista de la codificación es utilizando que el tipo p está habitado. Esto se explica diciendo que si p es un tipo vacío entonces p es falso, en cambio, si p tiene un elemento entonces p es verdadero. Por lo que el f que hemos definido antes nos determina que p es cierto.

Entonces si tenemos p : **Prop** y tomamos dos elementos f_1 $f_2 : p$, Lean va a entender estos dos elementos como idénticos. Esto se conoce como *proof irrelevance*, que se traduce en que no importa quienes sean las pruebas, únicamente representan objetos de la teoría de tipos dependientes que expresan que p es una proposición que es cierta. Este resultado está implementado en Lean bajo el nombre de **proof_irrel**:

```
theorem proof_irrel{a : Prop} (h₁ : a) (h₂ : a) :
    h₁ = h₂
```

3.2. Lógica proposicional en Lean

En Lean se definen los conectores de la lógica proposicional para enlazar correctamente los elementos del tipo **Prop**, esto se puede ver en el capítulo 3 de [2] (*Propositional Logic*). Para ello se tienen los caracteres para la negación, conjunción, disyunción, implicación y biimplicación que en Lean se verán como indica el ejemplo dónde se comentarán los accesos directos a dichos símbolos:

```
variable (p q : Prop)
#check p ∧ q --\and
#check p ∨ q --\or
#check p → q --\imp
#check p ↔ q --\iff
```

Es importante recordar que $\rightarrow$ (al igual que el resto de conectores) se asocia por la derecha y sigue teniendo la misma funcionalidad que cuando los elementos eran de un **Type*** en general en vez de **Prop**. Vamos a ver de que manera podemos utilizar estas conectivas en Lean para hacer demostraciones de proposiciones simples.

3.2.1. Conjunción y Disyunción

En el momento de tener una conjunción en Lean hay que hacer notar que estas expresiones tienen dos partes, la izquierda y la derecha. Es por ello que se utilizarán las reglas de eliminación **And.left** y **And.right** para crear pruebas de cada lado de la conjunción. Del mismo modo la regla de introducción, **And.intro** creamos una prueba de la conjunción.

Para entender esto correctamente vamos a hacer un ejemplo en Lean donde p y q son variables de tipo **Prop** y $p \wedge q$ es su conjunción también de tipo **Prop**.

Ejemplo 3.2.1 (Prueba de p y q)**.**

```
--Vamos a asumir en primer lugar que p y q son dos objetos de tipo Prop

variable (p q : Prop)

--Suponiendo que tengo dos hipotesis hp = "p es cierto" y
--hq = "q es cierto"

--La prueba de que p → q → p ∧ q
example (hp : p) (hq : q) : p ∧ q := by
  apply And.intro --Creamos 2 metas, una que es p y la otra q
  exact hp
  exact hq

--La prueba de que p ∧ q → p
example (h : p ∧ q) : p := by
--Extraigo el lado izquierdo de h que afirma que p es cierta
  exact And.left h

--La prueba de que p ∧ q → q
example (h : p ∧ q) : q := by
--Extraigo el lado izquierdo de h que afirma que p es cierta
  exact And.right h
```

En el ejemplo empezamos declarando dos variables de tipo proposicionales. Ahora vemos el uso de **apply** y **exact**, por ahora vamos a explicar la prueba desde el punto de vista de aplicar las reglas desde el punto de vista lógico, no obstante las tácticas y el modo táctico serán explicadas en el capítulo 4.

- En el primer **example** vamos a probar que dadas las hipótesis de que p y q son ciertas entonces $p \wedge q$ es cierta. Para ello se utiliza **And.intro** que crea dos nuevas metas, una que es p y la otra que es q por lo que utilizando las hipótesis de que son ciertas se resuelven dichas metas.
- En el segundo tenemos como hipótesis que $p \wedge q$ es cierta entonces p es cierta. Ahora tomamos el lado izquierdo de la hipótesis que cierra la meta.
- Por último repetimos el caso anterior para el lado derecho para probar que q es cierta a partir de la misma hipótesis $p \wedge q$.

Hay varias formas alternativas de escribir la prueba que acabamos de hacer para aligerar notación. Una de ellas es utilizar constructores anónimos de manera que cuando el tipo importante sea inductivo y se pueda inferir del contexto. La otra es utilizar que cuando una expresión es de tipo inductivo se puede utilizar su forma abreviada.

Vamos a ver un ejemplo de cada uno en este contexto en particular para arrojar luz a la idea que se acaba de exponer:

Ejemplo 3.2.2 (Constructores anónimos y abreviaturas de tipos inductivos)**.**

```
variable(p q : Prop)

--Con constructores anonimos podemos escribir una expresion
--equivalente a And.intro hp hq
example (hp : p) (hq : q) : p ∧ q := ⟨hp, hq⟩

--Vamos a utilizar que h.side es la abreviatura de And.side h
example (h : p ∧ q) : p := h.left
example (h : p ∧ q) : q := h.right
```

Utilizamos el constructor anónimo $\langle \cdot, \ldots, \cdot \rangle$ en el que básicamente ponemos el par de pruebas necesarias para cada lado de la conjunción. Este constructor es útil en otros contextos pero vamos a explicar solo su utilidad en este contexto. Lo bueno de esta notación es utilizarla cuando las expresiones de tipo conjuntivo son más extensas. En estas es donde se ve la verdadera utilidad de aligerar la notación.

Observación 3.2.3. Los constructores anónimos asocian por la derecha del mismo modo que las expresiones de tipo conjuntivo.

```
example (h : p ∧ q) : q ∧ p ∧ q :=
  ⟨h.right, h.left, h.right⟩

example (h : p ∧ q) : q ∧ p ∧ q :=
  ⟨h.right,⟨h.left, h.right⟩⟩
```

En el ejemplo, ambas demostraciones son equivalentes. Es lógico pues sabemos que en lógica de primer orden $q \wedge p \wedge q = q \wedge (p \wedge q)$, luego los constructores mantienen la asociatividad por la derecha como habíamos indicado.

En el caso de la disyunción es un poco diferente por la propia naturaleza de la misma. Para probar que $p \vee q$ es cierta basta con que sea cierta una de las dos. Es por ello por lo que en Lean podemos crear una prueba si tenemos como hipótesis que una de las dos proposiciones es cierta. Para ello existen las reglas de introducción **Or.intro_lado** que crean una prueba de la disyunción a partir de la hipótesis de que la proposición en el lado correspondiente es cierta. Veamos un ejemplo y sobre él se explicará el funcionamiento de la instrucción:

Ejemplo 3.2.4 (Prueba de $p \vee q$)**.**

```
variable(p q : Prop)

example (hp : p) : p ∨ q := Or.intro_left q hp
example (hq : q) : p ∨ q := Or.intro_right p hq
```

Observación 3.2.5. La necesidad del primer argumento viene dada por la construcción del tipo inductivo **Or**. Se construye utilizando ambos argumentos aunque solo necesitemos que uno sea cierto, para esclarecer esta idea veamos la definición que da Lean:

```
inductive Or (a b : Prop) : Prop where
  /--If ‘h : a‘ then ‘Or.inl h : a ∨ b‘. -/
  | inl (h : a) : Or a b
  /--If ‘h : b‘ then ‘Or.inr h : a ∨ b‘. -/
  | inr (h : b) : Or a b
```

Para evitar tener que escribir de forma explícita el argumento Lean nos ofrece dos facilidades. Una es clara ya que podemos utilizar directamente la definición del tipo inductivo, es decir, **Or.inl hp** u **Or.inr hq**. La segunda es aplicando la idea de inferir el primer argumento de forma anónima utilizando "_". Todas estas expresiones son equivalentes por lo que utilizaremos por conveniencia la más abreviada.

Al igual que en la conjunción queremos probar que una expresión se puede demostrar a partir de la disyunción. Para ello introducimos la relga de eliminación, **Or.elim**. Esta consiste en que una proposición es cierta a partir de $p \vee q$ mostrando que se sigue de p y se sigue de q. Es por ello que vamos a necesitar tres hipótesis para esta regla. Por lo que estas serán $h_1 : p \vee q$, $h_2 : p \to q \vee p$ y $h_3 : q \to q \vee p$. Veamos que a partir de $p \vee q$ deducimos $q \vee p$.

Ejemplo 3.2.6 (Prueba de $q \vee p$)**.**

```
variable(p q : Prop)

--Prueba utilizando tacticas
example(h : p ∨ q) : q ∨ p := by
  apply Or.elim h
  . exact Or.inr
    exact Or.inl

--Prueba utilizando funciones
example(h : p ∨ q) : q ∨ p :=
  Or.elim h (fun hp ↦ Or.inr hp) (fun hq ↦ Or.inl hq)
```

Es importante observar que Lean está infiriendo los tipos de hp y hq por contexto y que al igual que en la conjunción podemos abreviar la expresión mediante **h.elim**. No obstante si quisiésemos demostrar $q \vee p \vee q$, tendríamos que acceder a los campos del **Or** recordando que asocia por la derecha. Por lo tanto para $p \to q \vee p \vee q$ utilizamos la siguiente función **fun hp** $\mapsto$ **Or.inr (Or.inl hp)** y de forma análoga para q aunque en este caso tenemos dos opciones correctas. Con esto hemos abordado las posibilidades de estos conectores lógicos para ser utilizados en las demostraciones.

3.2.2. Negación

El siguiente elemento importante de la lógica que podemos utilizar en Lean es la negación que no deja de ser el hecho de que una proposición no es cierta. Es por ello por lo que dada una proposición p la negación en Lean se puede expresar como $p \to False$.

El objetivo de las demostraciones donde se utiliza la negación es buscar contradicciones. Una forma de hacer esto es utilizando **False.elim** que representa el hecho de que cualquier cosa se sigue de una contradicción. Por ejemplo, sabemos que si p y $\neg p$ son ciertos, podemos derivar de ellos cualquier proposición utilizando el principio de explosión, que en Lean es esta regla que acabamos de ver.

Internamente en Lean al aplicar **False.elim** se busca derivar *False* a partir de las hipótesis. Por lo que aplicando ahora $\neg p$ que es equivalente a $p \to False$ tendríamos como meta p que es una hipótesis, alcanzando así la prueba que buscamos.

Otra manera de hacer esto mismo es mediante la regla **absurd**. En este caso al aplicarse esta regla se generan dos metas, la primera será una proposición y la segunda esa misma proposición negada. Vamos a ver un ejemplo para entender el funcionamiento:

Ejemplo 3.2.7 (Ejemplo *ex falso quodlibet*)**.**

```
--Como q es cierto y q → p entonces p es cierto. A la vez vemos
--que p no es cierto luego podemos derivar diciendo hp : (hqp hq)
example (hnp : ¬p) (hq : q) (hqp : q → p) : r :=by
  apply absurd
  apply (hqp hq) --hp
  apply hnp
```

La prueba empieza con **absurd** generando dos metas, la primera espera la prueba de que una proposición es cierta, en este caso p; y la segunda espera esa proposición negada, que como vemos es $\neg p$. En resumen estamos probando que de algo que es falso podemos derivar cualquier cosa, en este caso r. En lógica este principio se llama *ex falso quodlibet*.

Observación 3.2.8. Más adelante veremos que para incluir como hipótesis los cuantificadores universales utilizaremos la táctica **intro**. Con esto podemos hacer que si la meta es $\neg p$, como $p \to$ *False* entonces la meta sería False añadiendo como hipótesis p.

3.2.3. Implicación y Bi-Implicación, Modus Ponens y Equivalencia Lógica

En un primer lugar sabemos que $p \leftrightarrow q$ es equivalente a $p \to q$ y $q \to p$. Para utilizar esto en Lean vamos a aplicar diferentes tácticas.

La regla de introducción **Iff.intro $\mathbf{h_1}$ $\mathbf{h_2}$** genera una prueba de que $p \leftrightarrow q$ a partir de las hipótesis $h_1 : p \to q$ y $h_2 : q \to p$. Para el otro lado de la implicación tenemos las reglas **Iff.mp h** e **Iff.mpr h** que generan pruebas de $p \to q$ y $q \to p$ a partir de $h : p \leftrightarrow q$.

Observación 3.2.9. En Lean **.mp** y **.mpr** son las siglas de *Modus Ponens* y *Modus Ponens Reversed* respectivamente.

Vamos a ver un ejemplo donde probamos que la conjunción es conmutativa utilizando la doble implicación. En primer lugar definimos un teorema que prueba este hecho y utilizando la introducción se generan metas para ambos lados de la implicación. Las metas se resuelven utilizando las reglas vistas en la parte de demostraciones utilizando la conjunción. Por simplicidad se muestra la prueba equivalente con constructores anónimos poniendo de nuevo de manifiesto la irrelevancia de la prueba. Por último utilizamos el teorema que hemos construido para construir mediante el *Modus Ponens* una prueba de que $p \wedge q \to q \wedge p$ y usando el *Modus Ponens Reversed* una prueba de que $q \wedge p \to p \wedge q$.

Ejemplo 3.2.10 (La conjunción es conmutativa)**.**

```
--Teorema demostrado con usando tacticas con apply
theorem and_comm₁ : p ∧ q ↔ q ∧ p := by
  apply Iff.intro
  . apply fun h : p ∧ q => And.intro (h.right) (h.left)
  apply fun h : q ∧ p => And.intro (h.right) (h.left)

--Teorema demostrado con constructores anonimos
theorem an_and_comm₁ : p ∧ q ↔ q ∧ p :=
  ⟨fun h : p ∧ q => ⟨h.right, h.left⟩, fun h : q ∧ p => ⟨h.right, h.left⟩⟩

--Modus Ponens
example (h : p ∧ q) : q ∧ p := by
  apply (and_comm₁ p q).mp
  exact h

--Modus Ponens Reversed
example (h : q ∧ p) : p ∧ q := by
  apply (and_comm₁ p q).mpr
  exact h
```

3.3. Construcciones utilizadas

En esta sección vamos a explicar las construcciones que vamos a utilizar en Lean para que se comprendan cuando se utilicen. Son de las más utilizadas en pruebas en Lean pero vamos a dejar por el camino muchas importantes al no ser necesarias en el estudio que estamos llevando a cabo.

3.3.1. Unión Indexada

Para poder utilizar uniones de familias de conjuntos en Lean se define **iUnion** como la unión de una familia indexada de conjuntos. Dada la familia indexada de conjuntos (s : I → Set α)[1] la definición de **iUnion** nos devuelve ($\bigcup$ i, s i : Set α) con (i : I).

Un teorema útil es **mem_iUnion** el cual asegura que un elemento pertenece a la unión indexada si y sólo si existe un índice i, para el cual es elemento pertenece a (s i). Este resultado va a resultar fundamental más adelante en el Capítulo 6. Justo debajo vamos su implementación en Lean.

```
theorem Set.mem_iUnion{α : Type u} {ι : Sort v} {x : α} {s : ι → Set α} :
    x ∈ ⋃ (i : ι), s i ↔ ∃ (i : ι), x ∈ s i
```

3.3.2. Estructuras

Del mismo modo que vimos anteriormente vamos a utilizar la palabra reservada **structure** para crear estructuras que describan los diferentes campos de un elemento que buscamos crear. Estas estructuras pueden ser muy simples, como veremos a continuación, o complejas como sería

[1] El tipo de I es **Sort*** y el de α es **Type***

la definición de un grupo con todas las propiedades que este requiere. Vamos a comentar algunas propiedades sobre las estructuras:

- **Definición de datos**: Las estructuras sirven para definir tipos de datos complejos compuestos de varios componentes.
- **Inmutabilidad**: Los campos de una estructura son inmutables, es decir, no pueden ser modificados después de crear un elemento de dicha estructura.
- **Herencia**: Lean soporta un tipo de herencia simple a través de extensiones de estructuras, donde una estructura puede extender otra añadiendo más campos.

Para entender la utilidad de las estructuras vamos a definir los puntos en $\mathbb{R}^3$ y la propiedad de sumar dos puntos.

```
structure punto3D (α : Type*)[Add α] :=
    (x : α)
    (y : α)
    (z : α)

def add_punto {α : Type*}[Add α](p q : punto3D α) : punto3D α :=
    {x := p.x + q.x, y := p.y + q.y , z:= p.x + q.z}
```

Observación 3.3.1. El uso de **Add** α representa que el tipo α esta restringido a los que tengan una instancia de la clase **Add**, lo que garantiza la posibilidad de sumar elementos de tipo α. Ahora veremos que significan las clases e instancias de las mismas.

3.3.3. Clases e instancias

Los *type class* que a partir de ahora vamos a llamar clases, son una forma de describir un conjunto de operaciones o propiedades que deben ser implementadas por un tipo. Tienen similitudes con las estructuras pero veamos en detalle sus características para resaltar sus diferencias:

- **Polimorfismo ad-hoc**: Las clases permiten definir funciones que pueden operar sobre diferentes tipos de datos de manera polimórfica.
- **Instancias**: Los tipos que implementan una clase deben proporcionar instancias de esa clase. Una instancia define cómo las operaciones de la clase se aplican a ese tipo. **Resolución implícita**: Lean puede resolver automáticamente qué instancia usar en un contexto dado, lo que permite la sobrecarga de funciones sin necesidad de especificar explícitamente el tipo en cada uso.

Un ejemplo simple de uso es la clase **Semigroup**. Esta clase implementa la estructura algebraica de semigrupo que es una estructura algebraica que consiste en un conjunto cerrado bajo una operación binaria asociativa. En Lean **Semigroup** es el caso particular del producto:

```
class Semigroup (G : Type u) extends Mul G where
    protected mul_assoc : ∀ a b c : G, a * b * c = a * (b * c)
```

Antes de continuar es importante destacar que el identificador de **protected** nos permite que el identificador que le sigue no se confunda con otros con el mismo nombre. Ahora vamos a crear una instancia de la clase **Semigroup** para los naturales. Para esto simplemente tenemos que definir la asocitividad del producto en este conjunto, que en Lean corresponde con el teorema **mul_assoc** dentro de **Nat**, veámoslo:

```
instance semigrupo : Semigroup ℕ where
    mul_assoc := Nat.mul_assoc
```

Capítulo 4

Tácticas, teoremas y construcciones importantes

Durante el capítulo anterior hemos estado utilizando palabras reservadas como **apply** y **exact**. Las demostraciones de este estilo decimos que se hacen en "Modo Táctico", estas utilizarán diferentes expresiones que se denominan **tácticas** de las que vamos a ir comentando las más relevantes de cara a ser utilizadas en pruebas de posteriores capítulos. La información relativa a estas tácticas vienen de la librería Matlib4.

4.1. Modo táctico, apply y exact

El modo táctico es una estrategia que tiene Lean para facilitar la formalización de pruebas en base a transformar la meta actual en una o varias metas más sencillas para llegar al estado *no goals* que nos quiere transmitir que la prueba es correcta.

Iniciar una demostración en este modo utilizaremos la palabra reservada **by** después de la definición del teorema. El bloque de tácticas separará la aplicación de las mismas bien con puntos y coma o bien con saltos de línea. Por comodidad vamos a utilizar siempre los saltos de línea, así es más fácil leer las pruebas.

Comprobemos la diferencia entre el uso de **apply** y **exact**. En primer lugar, **apply** ***táctica*** unifica la conclusión de ***táctica*** con la expresión del objetivo actual y crea nuevos objetivos para los argumentos restantes, siempre que ningún argumento posterior dependa de ellos. En segundo lugar, **exact** ***táctica*** es una variante de **apply** la cual indica que la expresión dada debe unificar exactamente sin generar nuevos objetivos. Es importante destacar que sin usar **exact** las pruebas no serán incorrectas pero resulta más robusto por su construcción puesto que se utiliza el tipo esperado al procesar la expresión que se pretende aplicar. Vamos a recuperar un ejemplo sencillo que hemos hecho en el capítulo previo:

```
variable (p q : Prop)

example (hp : p) (hq : q) : p ∧ q := by
  apply And.intro --Creamos 2 metas, una que es p y la otra q
  . exact hp
  exact hq
```

En este ejemplo vemos como utilizando **apply** se generan dos metas que resolvemos con su respectivo **exact**. Es importante comentar que utilizamos el punto para distinguir que estamos en un caso o en otro, es una manera de hacer más legible el código y saber que estamos haciendo en todo momento. También cabe destacar que las expresiones que son aceptadas por **apply** o **exact** pueden ser compuestas, simplificando la escritura del código aunque se pierda la perspectiva que ofrece el hacer las tácticas paso a paso.

```
variable (p q : Prop)
--Esta expresion es equivalente a la que acabamos de ver
example (hp : p) (hq : q) : p ∧ q := by
    exact And.intro hp hq
```

4.2. Tácticas básicas y resultados importantes

A parte de las tácticas que acabamos de ver hay otras muy útiles que vamos a comentar ya que serán necesarias para las pruebas posteriores. Vamos a definir de forma rápida una táctica única que sirve para dejar incompletas partes de una demostración. manteniendo un esqueleto de prueba sintácticamente correcto. Esta táctica se llama **sorry** y la utilizaremos siempre que haya una parte de una prueba que no va a ser probada pero no queremos que el código nos de errores.

4.2.1. Intro

La táctica **intro** es utilizada para introducir una hipótesis o variables de cualquier tipo. Esta va relacionada con los cuantificadores universales, es decir, que cuando tenemos expresiones con un para todo, ya sea explícito o implícito, mediante esta táctica se añadirá como hipótesis. Veamos un ejemplo y sobre él expliquemos el uso de **intro**.

```
example : ∀ a b c : Nat, a = b → a = c → c = b := by
  intro a b c h1 h2
  sorry
```

En el ejemplo utilizamos **intro** para añadir a las hipótesis las variables cualificadas existencialmente de forma explícita y luego las hipótesis $a = b$ y $a = c$. Por lo que vimos a la hora de definir los teorema, sabemos que el operador flecha va enlazando hipótesis hasta llegar a la conclusión. De esta manera **intro** va a añadiendo hipótesis hasta que solo queda el objetivo de la prueba. Utilizamos la palabra reservada **sorry** ya que utilizaremos una táctica posterior para terminar esta prueba.

Observación 4.2.1. El uso de la táctica **intros** permite la misma funcionalidad pero sin necesidad de ningún argumento previo, es decir, añade todas las variables e hipótesis que puede asignándoles un nombre cualquiera.

4.2.2. Obtain y use

La táctica **obtain** permite desempaquetar la información de los cuantificadores existenciales, es decir, añade una variable y una hipótesis nueva con la propiedad para la que existe dicha variable. Por otro lado **use** sirve para proporcionar un objeto determinado sobre el que se va a realizar la prueba, este lo usamos cuando en la meta tenemos un cuantificador existencial. Vamos a ver un ejemplo dónde usamos ambas tácticas:

```
def FnUb (f : ℝ → ℝ) (a : ℝ) := ∀ x, f x ≤ a
def FnHasUb (f : ℝ → ℝ) := ∃ a, FnUb f a

theorem fnUb_add {f g : ℝ → ℝ} { a b : ℝ} (hfa : FnUb f a) (hgb : FnUb g b) : FnUb
(fun x ↦ f x + g x) (a + b) := fun x ↦ add_le_add (hfa x) (hgb x)

example (ubf : FnHasUb f) (ubg : FnHasUb g) : FnHasUb fun x ↦ f x + g x := by
  obtain ⟨a, ubfa⟩ := ubf
  obtain ⟨b, ubgb⟩ := ubg
  use a + b
  apply fnUb_add
  . exact ubfa
  exact ubgb
```

En este ejemplo (extraído de la referencia [1]) estamos definiendo el concepto de que una función está acotada superiormente. Una vez teniendo los elementos básicos demostramos el teorema el cual afirma que dadas dos funciones acotadas superiormente entonces su suma también es una función acotada superiormente.

Con estos primeros resultados llegamos a la parte donde usamos las tácticas que estamos viendo. Se pretende probar que dadas dos funciones para las que tenemos sus respectivas cotas superiores entonces podemos obtener la cota superior de la suma de ellas. Para proceder utilizamos **obtain** de modo que se crean las variables **a** y **b** así como las hipótesis de que **a** es la cota superior de **f** y **b** es la cota superior de **g**, siendo estas hipótesis, **ubfa** y **ubgb** respectivamente.

Una vez desempaquetados los cuantificadores existenciales utilizamos el **use** para proporcionar $\mathbf{a + b}$ como el objeto que debe usar el cuantificador existencial de la meta para llegar a que $\mathbf{f + g}$ tiene una cota superior. Lo que resta de la prueba es ver que $\mathbf{f\ x + g\ x\ \leq a + b}$ basándose en que $\mathbf{f\ x \leq a}$ y $\mathbf{g\ x \leq b}$.

4.2.3. Rewrite

La táctica **rewrite** que escribiremos como **rw** nos es de utilidad para aplicar sustituciones a objetivos e hipótesis. En el caso de utilizar **rw[exp]** haremos la sustitución directamente en la meta, si queremos hacer una sustitución en una hipótesis la forma será **rw[exp] at h**. Vamos a ver el ejemplo (extraído de la referencia [1]) del apartado anterior que dejamos sin demostrar:

```
example : ∀ a b c : Nat, a = b → a = c → c = b := by
  intro a b c h1 h2
  rw[← h2]
  exact h1
```

En esta prueba hacemos un **rewrite** sobre la meta pero podemos observar que tenemos una flecha que representa que la sustitución de la expresión en orden inverso al natural. En este caso particular lo que hacemos es añadir las variables **a**, **b** y **c** y las hipótesis **a** = **b** y **a** = **c**. Por lo tanto la sustitución que se hace modifica la meta **c** = **b** en **a** = **b**, si llegamos a hacer el **rewrite** por defecto Lean nos informaría de que la expresión no unifica.

4.2.4. Simplify

La táctica **simplify** que escribiremos como **simp** nos permite hacer sustituciones a objetivos e hipótesis de manera automática para llegar a una expresión, en teoría, más simple. Para hacer esto posible se etiquetan expresiones de Lean como [**simp**] para, al utilizar la táctica **simp**, poder ir reescribiendo los términos en cadena hasta llegar a la expresión lo más simple posible.

```
example (a b : ℕ) (h : a ≤ b) : a * 1 ≤ 1 * b := by
  simp
  exact h
```

Este ejemplo es muy simple pero podemos observar que gracias a **simp** podemos eliminar los productos por uno. Es importante decir que es capaz de simplificar expresiones mucho más complejas, de hecho lo veremos más adelante en demostraciones más extensas.

4.2.5. Refine'

Esta táctica permite construir una prueba de manera estructurada y detallada. Un ejemplo sencillo para comprender su utilización, es afirmar que existe una función natural de modo que para cada natural su imagen es su sucesor. En este caso **refine'** se define con un par donde el primer campo es un elemento de los que se busca y el segundo campo es una prueba de que cumple la propiedad.

```
example : ∃ f : ℕ → ℕ , ∀ n : ℕ, f n = n + 1 := by
    refine' ⟨_,_⟩
    . use fun i => i + 1
    intro n
    exact rfl
```

Otra versión de esta táctica es **refine** que es más restrictiva, no obstante en las pruebas posteriores utilizaremos **refine'** para hacer pruebas de tipo existencial.

4.2.6. Choose y Choose_spec

Estos resultados ayudan a resolver enunciados de tipo existencial. En el caso de **choose** si tenemos una hipótesis del tipo (h : ∃ x, P x) entonces al aplicar **choose** h se nos devuelve el valor del x que cumple (P x). Por otro lado **choose_spec** es un teorema que dada la hipótesis h afirma que p se cumple para el elemento elegido por **choose** h. Veamos un ejemplo muy sencillo del uso de esta táctica:

```
open Classical

variable {S : Type*}[Semigroup S]

noncomputable def weak_inv (h : ∀ (x : S), ∃! y, x * y * x = x) : S → S := fun x =>
choose (h x)

lemma weak_inv_prop (h : ∀ (x : S), ∃! y, x * y * x = x) :
 ∀ x, x * weak_inv h x * x = x := by
    intro x
    exact (choose_spec (h x)).1
```

En este ejemplo declaramos un tipo de un semigrupo, recordemos que en Lean corresponde con un tipo en el que se cumple que la multiplicación es asociativa, y creamos una definición y un lema:

- La definición se etiqueta como **noncomputable** porque se está utilizando un principio no constructivo de elección como es **choose**. Lo que se define es la propiedad de que solo hay un elemento y de modo que $x \cdot y \cdot x = x$, es una propiedad de elemento inverso pero más débil. Esta definición se crea eligiendo un elemento con **choose**.

- La propiedad de existencia de un elemento inverso débil tiene como hipótesis que para todo elemento del semigrupo existe un único elemento inverso débil y queremos probar que el elemento escogido por **choose** es, en efecto, ese único elemento. Para lograr esto utilizamos **choose_spec** para escoger la propiedad necesaria para acabar la prueba.

Para terminar con el ejemplo hay que comentar dos detalles extra. La primera es que hay que abrir el espacio de nombres **Classical**, un conjunto de definiciones y teoremas que para ser utilizados hay que abrir este espacio. La segunda es entender porqué **choose_spec** parece ser un tipo conjuntivo. Vamos a escribir el tipo que tiene y con esto habremos terminado el ejemplo:

```
(fun y => x * y * x = x) (choose (_ : ∃! y, x * y * x = x)) ∧
∀ (y : S), (fun y => x * y * x = x) y → y = choose (_ : ∃! y, x * y * x = x) : Prop
```

Viendo este tipo nos queda claro que lo que necesitamos para demostrar el lema es la parte izquierda de la conjunción donde aplica el elemento escogido por **choose** a la propiedad que pretendemos demostrar.

Capítulo 5

Axioma de elección, Lema de Zorn y principio de buena ordenación

En este capítulo vamos a ver la prueba de que el Axioma de Elección, el Lema de Zorn y el Principio de Buena Ordenación son equivalentes. Vamos a ir dando las definiciones necesarias para poder probar esta equivalencia y una vez conseguido este objetivo pasaremos a formalizar en Lean este resultado.

5.1. Definiciones previas y resultados importantes

Antes de hacer la prueba de la equivalencia vamos a definir conceptos importantes y poner algunos resultados relacionados con ellos que puedan resultar de interés.

Definición 5.1.1 (Axioma de elección, **AE**)**.** Sea una familia de conjuntos $\{X_i\}_{i\in I}$ con $X_i \neq \emptyset$ $\forall i \in I$, entonces existe una función de elección $s : \{X_i\}_{i\in I} \to \bigcup_{i\in I} X_i$ de manera que $s(X_i) \in X_i$ $\forall i \in I$.

El axioma de elección ayuda a probar algunos resultados conocidos. Vamos a comentar dos de los más importantes y el hecho esencial de utilizar del axioma de elección en las demostraciones de ellos.

- **Modelo de Solovay y teorema de Vitali**:

  Robert Martin Solovay demostró que la existencia de un conjunto no medible Lebesgue no se puede probar en la teoría de conjuntos de Zermelo-Fraenkel, a partir de ahora ZF. La axiomática de Zermelo-Fraenkel, ZF, permite definir la teoría de conjuntos sin utilizar el axioma de elección, la variante que sí utiliza este resultado se abrevia con ZFC por la palabra *choice*. El modelo de Solovay [**?**] es un modelo de ZF en el se cumple el axioma de elección numerable, en el que todo conjunto es medible Lebesgue y en el que el axioma de elección completo no se cumple. Esto nos lleva a que para que los conjuntos no medibles existan sería necesario asumir

el axioma de elección. Podemos confirmar esto con el teorema de Vitali [11] que demuestra la existencia de conjuntos de número reales que no son medible lebesgue. La prueba se hace utilizando el axioma de elección y no hay manera de probar este resultado sin asumirlo.

- **Paradoja de Banach-Tarski**

  El teorema de Banach-Tarski [9] es un resultado de teoría de conjuntos que demuestra que se puede dividir $S^2 = \{x \in \mathbb{R}^3 : \|x\| = 1\}$ en ocho partes disjuntas dos a dos tal que, en base a giros en $\mathbb{R}^3$, pueden reagruparse en dos copias idénticas al conjunto original, es decir, dos veces el conjunto S^2. Como los 8 fragmentos no son medibles no se puede argumentar que la medida se preserva por rotaciones, entonces el uso del axioma de elección en la demostración consigue que se pueda terminar sin problema.

Después de exponer el Axioma de Elección y su utilidad vamos a ver el Lema de Zorn y algunos ejemplos en los cuales resulta útil. Antes de pasar al resultado necesitamos definir el concepto de conjunto inductivo.

Definición 5.1.2. Un conjunto parcialmente ordenado $(X; R)$ es inductivo si toda cadena tiene cota superior.

Observación 5.1.3. Recordemos que una cadena es un subconjunto totalmente ordenado.

Definición 5.1.4 (Lema de Zorn, **LZ**)**.** Todo conjunto inductivo tiene un elemento maximal.

El Lema de Zorn es un resultado que es necesario en la demostración de múltiples resultados. Los vemos a continuación con sus respectivas pruebas:

Teorema 5.1.5. *Sea V un k-espacio vectorial y $(X, \subseteq)$ la familia de conjuntos cuyos elementos son linealmente independientes. Entonces, existe una base B de V que contiene a X*

Demostración. Sea C una cadena de X. Si definimos $S = \bigcup_{A \in C} A$ veamos que es linealmente independiente ya que sean $\alpha_1, \ldots, \alpha_n \in K$ y $s_1, \ldots, s_n \in S$ si $\alpha_1 \cdot s_1 + \cdots + \alpha_n \cdot s_n = 0$ sabemos que cada $s_i \in A_i$ y por construcción sabemos que existe un $A_j \in C$ de modo que $s_i \in A_j$ para $1 \leq i \leq n$. Ahora al tener que $C \subseteq X$ entonces $A_j \in X$ lo que nos dice que los $s_i \in A_j$ son linealmente independientes en K por lo que $a_i = 0$ para $1 \leq i \leq n$ llevándonos a ver que S es linealmente independiente. Ahora como hemos tomado una cadena cualquiera C que está acotada superiormente por un S llegamos a la conclusión de que $(X, \subseteq)$ es inductivo y por el Lema de Zorn sabemos que tiene un elemento maximal que va a ser la base que estamos buscando. □

Antes de continuar es necesario definir un concepto, la noetherianidad de un anillo. A partir de él vamos a ver diferentes resultados conocidos que lo utilizan para su prueba.

Definición 5.1.6 (Anillo noetheriano)**.** Un anillo A es noetheriano si dada una cadena numerable

$$\mathfrak{a}_1 \subseteq \mathfrak{a}_2 \subseteq \cdots \subseteq \mathfrak{a}_n \subseteq \ldots$$

en la familia de ideales de A existe $n \geq 1$ tal que $\mathfrak{a}_{n+k} = \mathfrak{a}_n$ para todo $k \geq 1$.

Observación 5.1.7. La condición anterior se conoce como la condición de cadena ascendente y el hecho de que a partir de cierto $n \geq 1$ todos los siguientes ideales son el mismo se dice que la cadena estaciona o es estacionaria. Por lo tanto un anillo noetheriano es aquel que cumple la condición de cadena ascendente.

Teorema 5.1.8. *Todo elemento no unidad de un anillo está contenido en un ideal maximal.*

Demostración. La familia de ideales

$$\mathcal{F} = \{\mathfrak{b} \subsetneq A \text{ ideal de } A : \mathfrak{a} \subset \mathfrak{b}\}$$

donde $\mathfrak{a}$ es un ideal propio de A y $\mathcal{F} \neq \emptyset$ ordenado parcialmente respecto de la inclusión.

Además, toda cadena $\mathcal{C} := \{\mathfrak{b}_i\}_{i\in I}$ de $\mathcal{F}$ está acotada superiormente ya que $\mathfrak{b} := \bigcup_{i\in I} \mathfrak{b}_i \in \mathcal{F}$ es una cota superior suya. Por el Lema de Zorn, existe un elemento maximal $\mathfrak{m} \in \mathcal{F}$ y vamos a ver que $\mathfrak{m}$ es un ideal maximal de A que contiene a $\mathfrak{a}$. Si $\mathfrak{m}$ no es un ideal maximal de A entonces existe un ideal $\mathfrak{b} \subsetneq A$ tal que $\mathfrak{m} \subsetneq \mathfrak{b}$. Pero entonces $\mathfrak{a} \subseteq \mathfrak{m} \subsetneq \mathfrak{b} \subsetneq A$ por lo que $\mathfrak{b} \in \mathcal{F}$. Contradiciendo así la maximalidad de $\mathfrak{m}$ como elemento de $\mathcal{F}$. Esta prueba es el apartado (1) de la Proposición I.2.10 de [6]. □

Antes de la siguiente proposición necesitamos definir dos conceptos previos que utilizaremos en el enunciado y en la prueba.

Definición 5.1.9. Un ideal $\mathfrak{a}$ de A es finitamente generado si es el ideal generado por un subconjunto finito $S := \{x_1, \ldots, x_r\} \subseteq A$. En tal caso $\mathfrak{a} = x_1A + \cdots + x_rA := \{a_1x_1 + \cdots + a_rx_r : a_i \in A\}$ y se suele denotar $\mathfrak{a} := (x_1, \ldots, x_r)$.

Definición 5.1.10. Un anillo A cumple la condición maximal si toda familia no vacía de ideales de A tiene algún elemento maximal con respecto de la inclusión.

Ahora con estas definiciones vamos a demostrar el siguiente resultado.

Proposición 5.1.11. *Sea A un anillo. Las siguientes afirmaciones son equivalentes:*

(1) Todos los ideales de A son finitamente generados.

(2) A es noetheriano..

(3) El anillo A cumple la condición maximal.

Demostración. (1) $\Longrightarrow$ (2) Sea una cadena de ideales $\mathfrak{a}_1 \subseteq \cdots \subseteq \mathfrak{a}_n \subseteq \ldots$ de A. Tenemos que $\mathfrak{a}$, que es la unión numerable de esta cadena, es un ideal de A, por lo que es finitamente generado. Si llamamos $\mathfrak{a} = (a_1 \ldots a_r)$ entonces para cada $1 \leq i \leq r$ tenemos un n_i de modo que $a_i \in \mathfrak{a}_{n_i}$ por lo que todos los generadores del ideal pertenecen al ideal $\mathfrak{a}_n$ donde n es el máximo de los n_i, esto implica para cada $k \geq 1$ se tiene $\mathfrak{a}_{n+k} \subseteq \mathfrak{a} = \mathfrak{a}_n \subseteq \mathfrak{a}_{n+k}$.

(2) $\Longrightarrow$ (3) Inmediato por el Lema de Zorn.

(3) $\Longrightarrow$ (1) Supongamos que existe un ideal $\mathfrak{a}$ de A no finitamente generado. Elegimos un elemento del ideal, a_1 y por lo que estamos suponiendo, $\mathfrak{a}_1 := a_1 A \subsetneq \mathfrak{a}$ esto nos lleva a tener un $a_2 \in \mathfrak{a} \setminus \mathfrak{a}_1$. Como $\mathfrak{a}$ no es finitamente generado, se tiene una inclusión estricta $\mathfrak{a}_2 := a_1 A + a_2 A \subsetneq \mathfrak{a}$. Siguiendo esta lógica obtenemos una sucesión de elementos de A de modo que,

$$\mathfrak{a}_1 \subsetneq \mathfrak{a}_2 \subsetneq \cdots \subsetneq \mathfrak{a}_n := a_1 A + \ldots a_n A \subsetneq \ldots$$

entonces tenemos una familia no vacía de ideales de A sin elementos maximales, contradiciendo la hipótesis. Esta prueba está basada en la Proposición I.2.5 de [6]. □

Teorema 5.1.12. *(**Nullstellensatz** o Teorema de la base de Hilbert): Sea A un anillo noetheriano. Entonces, el anillo de polinomio $A[t]$ es noetheriano.*

Demostración. Supongamos, por reducción al absurdo, que existe un ideal $\mathfrak{a}$ de $A[t]$ que no es finitamente generado. Elegimos un polinomio $f_1 \in \mathfrak{a} \setminus \{0\}$ tal que su grado es $\deg(f_1) = \min\{\deg(f) : f \in \mathfrak{a} \setminus \{0\}\}$ y construímos inductivamente los polinomios

$$f_{k+1} \in \mathfrak{a} \setminus (f_1, \ldots, f_k)A[t]$$

cuyo grado es minimos entre los grados de los polinomios de $\mathfrak{a} \setminus (f_1, \ldots, f_k)A[t]$. Sean $d_k := \deg(f_k)$ y a_k el coeficiente director de f_k. Nótese que $d_k \leq d_{k+1}$ si $k \geq 1$. Denotamos $\mathfrak{b}_k := (a_1, \ldots, a_k)A$ y observamos que $\mathfrak{b}_k \subseteq \mathfrak{b}_{k+1}$ para cada $k \geq 1$. Como A es noetheriano, existe un entero $n \geq 1$ tal que $\mathfrak{b}_n = \mathfrak{b}_{n+l}$ para cada $l \geq 1$.En particular, el coeficiente director de f_{n+1} debe pertenecer al ideal $\mathfrak{b}_n$. Eso implica que a_{n+1} se puede expresar como combinación lineal de $a_1, \ldots, a_n$. Es decir, existen $b_1, \ldots, b_n \in A$ tales que $a_{n+1} = \sum_{k=1}^{n} a_k b_k$. Entonces el polinomio,

$$g := f_{n+1} - \sum_{k=1}^{n} b_k f_k t^{d_{n+1}-d_k} \in \mathfrak{a} \setminus (f_1, \ldots, f_n)A[t] \quad \& \quad \deg(g) < \deg(f_{n+1}),$$

lo que contradice la elección de f_{n+1}. Por tanto, todo ideal de $A[t]$ es finitamente generado, es decir, noetheriano. La prueba está extraída del Teorema V.2.14 de [6]. □

Aunque el Lema de Zorn no se menciona explícitamente en la demostración del teorema de Hilbert, es fundamental que A cumpla la condición de la cadena ascendente y utilizamos el Lema de Zorn para probar la equivalancia de que todos los ideales de A son finitamente generados. De

todos modos hay pruebas del teorema en las que no se utiliza el Lema de Zorn, como en el capítulo 4 de [5].

Ahora para terminar necesitamos definir el Principio de Buena Ordenación para en la siguiente sección probar que el Axioma de Elección, el Lema de Zorn y el Principio de Buena Ordenación son equivalentes.

Definición 5.1.13 (Principio de Buena Ordenación, **PBO**)**.** Sea X un conjunto decimos que cumple el principio de buena ordenación si existe un buen orden en él de modo que $\forall A \subseteq X$ no vacío $\exists m \in A$ tal que $\forall n \in A$, $R\ m\ n$. Para entenderse más fácilmente lo que hacemos es decir que A tiene un elemento mínimo.

5.2. Prueba de la equivalencia

En esta sección vamos a abordar la prueba de la equivalencia de la siguiente manera: En primer lugar vamos a ver que el principio de buena ordenación implica el axioma de elección, en segundo lugar veremos que el Lema de Zorn implica el principio de buena ordenación y por último que el axioma de elección implica el Lema de Zorn inspirando la prueba del capítulo *Libertad de Elección* en [4].

Teorema 5.2.1. *El Principio de Buena Ordenación implica el Axioma de Elección.*

Demostración. Sea X un conjunto, $A \subseteq X$ no vacío y $\{A_i\}_{i\in I}$ una familia arbitraria de subconjuntos no vacíos de A. Por el **PBO** podemos escoger un buen orden R en X, de manera que A tiene un mínimo, denotado $\min(A)$. Tomando entonces que $\forall i \in I$, $A_i \subseteq X$ sabemos que existen $\min(A_i)$ para cada A_i. Por lo que basta con tomar una función de elección que envíe cada conjunto a su mínimo:

$$\begin{aligned} s : \{A_i\}_{i\in I} &\to \bigcup_{i\in I} A_i \\ A_i &\mapsto \min(A_i) \end{aligned}$$

Ahora como $\min(A_i) \in A_i$ se cumple el Axioma de Elección. □

Teorema 5.2.2. *El Lema de Zorn implica el Principio de Buena Ordenación*

Demostración. Sea un conjunto X y $\mathcal{F} = \{(A; R) : A \subseteq X, A \neq \emptyset \wedge R \text{ es un buen orden}\}$. Ahora si tomamos $(A_1; R_1)$, $(A_2; R_2) \in \mathcal{F}$ decimos que $(A_1; R_1) \leq (A_2; R_2)$ si:

(1) $A_1 \subseteq A_2$

(2) $R_2|_{A_1} = R_1$ y $\forall b \in A_2 \setminus A_1$ son posteriores en R_2 a los elementos de A_1

Sea ahora $\mathcal{C} = \{(A_i; R_i)\}_{i\in I}$ una cadena de $\mathcal{F}$ vamos a ver que (A;R) es una cota superior de $\mathcal{C}$, dónde:

(1) $A = \bigcup_{i \in I} A_i$

(2) aRb si aR_ib para algún $i \in I$

Con esto podemos ver que R es un buen orden, es decir, que es un orden total y tiene un elemento mínimo.

Para ver que es un orden total partimos de que cualquier R_i es un buen orden por definición. Como R es una extensión de cada R_i tenemos que aRb si aR_ib para algún i, por lo tanto $\forall a, b \in A$ tenemos que aRb, bRa o $a = b$ porque sabemos que $a, b \in A_i$ de modo que aR_ib o bR_ia o $a = b$.

Ahora tomamos $B \subseteq A$ y buscamos que B tenga un mínimo en R. Por construcción sabemos que existe un A_k de modo que $B \subseteq A_k$. Como R_k es un buen orden entonces sabemos que existe un $m_k \in B$ que es mínimo en R_k. Ahora como el orden relativo de B en R es el mismo que en R_k por extensión, llegamos a que existe un $m \in B$ que es mínimo en R que además cumple que $m = m_k$.

Entonces $A_i \subseteq A$ y $R|_{A_i} = R_i$ y $(A_i; R_i) \leq (A; R)$ por lo que utilizando el Lema de Zorn llegamos a que existe $(M; R)$ maximal. Basta ver que $M = X$ para llegar a la conclusión de que está bien ordenado.

Sea $x \in X \setminus M$ y completo M con $\bar{M} = M \cup \{x\}$ con un nuevo orden $\bar{R}$. Definamos este nuevo orden con precisión:

(1) Si $a, b \in M$ entonces $a\bar{R}b \iff aRb$

(2) Si $a \in M$ entonces $a\bar{R}x$

Por lo que x es el elemento más grande del orden y como $\bar{R}$ es un buen orden en $\bar{M}$ llegamos a que $(M; R) \leq (\bar{M}; \bar{R})$ contradiciendo el hecho de que $(M; R)$ sea maximal. Por lo que necesariamente $M = X$. □

Nos queda únicamente probar que el Axioma de Elección implica el Lema de Zorn. Para ello vamos a necesitar un resultado previo y varias definiciones, con los cuales la prueba resultará mucho más sencilla.

Lema 5.2.3. *Sea $\mathcal{F}$ una colección no vacía de subconjuntos de otro conjunto no vacío X. Hagamos varias suposiciones:*

(1) Para toda $\mathcal{C} \subseteq \mathcal{F}$ cadena, se cumple que $\bigcup_{c_i \in \mathcal{C}} c_i \in \mathcal{F}$.

(2) Sea $\varphi : \mathcal{F} \to \mathcal{F}$ tal que $A \subseteq \varphi(A)$.

(3) Se cumple que $\forall A \in \mathcal{F}$ el conjunto $\varphi(A) \setminus A$ contiene a lo sumo un elemento.

Entonces existe $A \in \mathcal{F}$ tal que $\varphi(A) = A$

Definición 5.2.4 (Tribu)**.** Sea $A_0 \in \mathcal{F}$ entonces diremos que $\mathcal{G} \subseteq \mathcal{F}$ es una tribu si:

(i) $A_0 \in \mathcal{G}$

(ii) Sea $\mathcal{C}$ una cadena en $\mathcal{G} \implies \bigcup_{c_i \in \mathcal{C}} c_i \in \mathcal{G}$

(iii) $A \in \mathcal{G} \implies \varphi(A) \in \mathcal{G}$

Definición 5.2.5 (Cronopio)**.** Siendo $\mathcal{F}_0$ la intersección de todas las tribus de una colección $\mathcal{F}$. Definimos Cronopio como un $C \in \mathcal{F}_0$ tal que $\forall A \in \mathcal{F}_0 \implies A \subseteq C \vee C \subseteq A$

Demostración. Veamos en primer lugar que los elementos de $\mathcal{F}$ que contienen a A_0 son una tribu. En efecto, sean $\mathcal{G} \subseteq \mathcal{F}$ tal que $\forall A \in \mathcal{G}, A_0 \subseteq A$, entonces:

(i) $A_0 \in \mathcal{G}$ pues $A_0 \subseteq G \in \mathcal{G}$

(ii) Sea una cadena $\mathcal{C} = \{c_i\}_{i \in I} \subseteq \mathcal{G}$ como sabemos que $\forall i \in I, c_i \in \mathcal{G} \implies \bigcup_{i \in I} c_i \in \mathcal{G}$

(iii) Sea $A \in \mathcal{G}$ como $A \subseteq \varphi(A)$ entonces $A_0 \subseteq \varphi(A) \implies \varphi(A) \in \mathcal{G}$

Tomemos ahora $\mathcal{F}_0$ la intersección de todas las tribus de $\mathcal{F}$, vemos que $\mathcal{F}_0$ es una tribu, de hecho, la más pequeña de todas. También veremos que es una cadena de $\mathcal{F}$.

Definamos ahora $\mathcal{K}$ como la familia de cronopios de $\mathcal{F}_0$. Dado ahora un cronopio $C \in \mathcal{K}$ definimos

$$\mathcal{F}_0(C) := \{A \in \mathcal{F}_0 : A \subseteq C \vee \varphi(C) \subseteq A\}$$

Si ahora $A \in \mathcal{F}_0$ entonces hay tres opciones:

(1) $A \subset C$ y $A \neq C$

(2) $A = C$

(3) $\varphi(C) = A$

- En (1) C no puede ser subconjunto propio de $\varphi(A)$ pues si $C \subsetneq \varphi(A)$ y $A \subseteq \varphi(A)$ tenemos que $\varphi(A) \setminus A$ tiene a lo sumo un elemento pero como por hipótesis $A \subsetneq C$ tendría mínimo dos elementos. Y como C es un cronopio llegamos a que $\varphi(A) \subseteq C$.
- En (2) se deriva que $\varphi(A) = \varphi(C)$.

- En (3) como $A \subseteq \varphi(A)$ llegamos a que $\varphi(C) \subseteq \varphi(A)$.

En resumen si $A \in \mathcal{F}_0(C)$ entonces como $\varphi(A) \subseteq C \vee \varphi(C) \subseteq \varphi(A) \implies \varphi(A) \in \mathcal{F}_0(C)$ por lo que llegamos a que $\mathcal{F}_0(C)$ es una tribu y como $\mathcal{F}_0(C) \subseteq \mathcal{F}_0$ y esta es la menor de todas entonces $\mathcal{F}_0(C) = \mathcal{F}\ \forall C \in \mathcal{K}$.

En otras palabras si $A \in \mathcal{F}_0$ y $C \in \mathcal{K}$ entonces $A \subseteq C \vee \varphi(C) \subseteq A$ con lo que $\varphi(C) \in \mathcal{K}$ y $\mathcal{K}$ es una tribu dentro de $\mathcal{F}_0$ lo que nos dice que $\mathcal{K} = \mathcal{F}_0$ y está totalmente ordenado por ser un conjunto de cronopios, esto nos dice que $\mathcal{F}_0$ es una cadena.

Tomando ahora $A = \bigcup_{A_i \in \mathcal{F}_0} A_i$ como sabemos que $A \in \mathcal{F}_0$ y $\varphi(A) \in \mathcal{F}_0$ pero como A es el mayor conjunto contenido en $\mathcal{F}_0$ y $A \subseteq \varphi(A)$ y $\varphi(A) \subseteq A$ entonces $A = \varphi(A)$ □

Teorema 5.2.6. *El Axioma de Elección implica el Lema de Zorn.*

Demostración. Sea $(X; R)$ inductivo y sea $\mathcal{F}$ la colección de todas las cadenas de X. Sabemos que $\mathcal{F} \neq \emptyset$ pues como mínimo para todo $x \in X$ tenemos la cadena $\{x\}$. Al unir conjuntos totalmente ordenados obtenemos conjuntos totalmente ordenados. Gracias al Axioma de Elección sabemos que existe una función s tal que $\forall A \subseteq X$ no vacío, $s(A) \in A$.

Tomemos una cadena $E \in \mathcal{F}$ y sea $E^* = \{x : x \in X \setminus E \wedge E \cup \{x\} \in \mathcal{F}\}$.

Si E^* es no vacío entonces $\varphi(E) = E \cup \{s(E^*)\}$ por lo contrario, de si serlo, entonces $\varphi(E) = E$.

Por el lema clave existe un $E \in \mathcal{F}$ tal que $\varphi(E) = E$, es decir, que E^* es vacío por lo que siguiendo la definición de E^* sabemos que E es una cadena maximal de elementos de X.

Como $(X; R)$ es inductivo entonces E tendrá una cota superior ub que es un elemento maximal en $(X; R)$. En caso contrario, existe $ub' \neq ub$ tal que $ubRub'$ pero entonces $\bar{E} = E \cup ub'$ sería una cadena estrictamente mayor, contradiciendo la maximalidad de E. □

Capítulo 6

Formalización en Lean

Una vez hemos visto como probar de forma clásica los teoremas vamos a pasar a formalizar, en primer lugar, los enunciados para luego mostrar sus respectivas pruebas. En Lean resulta fundamental una buena formalización de las definiciones ya que, en caso contrario, las pruebas resultarán imposibles de redactar.

6.1. Definición de variables globales

Antes de comenzar con las definiciones y enunciados vamos a definir las variables que vamos a ir utilizando en un futuro.

- Lo primero es utilizar un tipo global que va a ser sobre el que desarrollaremos todos los resultados posteriores, este tipo será γ. También necesitaremos un tipo que represente unos índices para hacer una familia indexada de conjuntos, este será I.
- En segundo lugar vamos a definir un conjunto S de γ y una familia de conjuntos del mismo tipo X.

```
--Definimos el tipo del que dependera el resto.
variable (γ : Type _)
--Para la familia de conjuntos de los X_i con i ∈ I
variable {I : Type _}
variable (X : I → Set γ)

variable (S : Set γ )
```

6.2. Definiciones y enunciados importantes

Ahora que tenemos las variables sobre las que vamos a trabajar vamos a crear definiciones en Lean que se correspondan con definiciones de conceptos importantes así como los enunciados de teoremas y lemas que serán utilizados.

- **Es un buen orden**: Esta definición va a tomar una relación tal que para todo conjunto del tipo Set γ existe un elemento que, bajo la relación dada, es mínimo.

```
def is_well_ordered (R : γ  → γ → Prop) : Prop := ∀ (A : Set γ),
Nonempty A → ∃ n ∈ A, ∀ m ∈ A, R n m
```

- **Principio de buena ordenación**: Utilizando la definición anterior, definimos el principio de buena ordenación como la existencia de una relación que es un buen orden en γ.

```
def well_ordered_principle : Prop := ∃ (R : γ  → γ → Prop),
is_well_ordered γ R
```

- **Axioma de elección**: Para esta definición se va a tomar que para todo familia indexada de conjuntos no vacíos existe una función de elección que envía elementos de esa familia a su unión haciendo que envíe cada elemento de la familia a sí mismo.

```
def axiom_of_choice : Prop := (∀ (i : I), Nonempty (X i)) →
∃ (f : (I → ⋃ i, X i)), ∀ (i : I), (f i).1 ∈ X i
```

- **Es cadena**: Tomando un orden parcial γ y c un Set γ, decimos que es una cadena si para todos dos elementos $x, y \in$ c se cumple que $x \leq y$ o $y \leq x$.

```
def is_chain [PartialOrder γ] (c : Set γ) : Prop := ∀ (x y : γ),
x ∈ c → y ∈ c → x ≤ y ∨ y ≤ x
```

- **Es inductivo**: Tomando un orden parcial γ y S un Set γ decimos que S es inductivo si para toda cadena en S existe una cota superior de dicha cadena.

```
def inductive_set [PartialOrder γ] (S : Set γ) : Prop := ∀ (c : Set γ),
c ⊆ S → is_chain γ c → ∃ (ub : γ), ∀ (x : γ ), x ∈ c → x ≤ ub
```

- **Lema de Zorn**: Dado un orden parcial γ y un conjunto inductivo S se afirma que existe un elemento de S que es maximal.

```
def zorn [PartialOrder γ] (_ : inductive_set S ) : Prop := ∃ (m : γ),
m ∈ S ∧ ∀ (x : γ), x ∈ S → x < m
```

6.3. Demostraciones basadas en los enunciados

Una vez hechas las definiciones pertinentes vamos a pasar a demostrar dos de las tres implicaciones necesarias para probar la equivalencia que ya hemos visto en el Capítulo 5. Vamos a utilizar la palabra reservada **sorry** en los resultados que no se prueben para que no de error en el código. Esto es útil también para poder utilizar resultados previos sin probarlos y dejar estas demostraciones para más adelante, es decir, podemos dejar un lema con **sorry** y demostrar el teorema en el que es necesario.

6.3.1. El Principio de Buena Ordenación implica el Axioma de elección

Para esta prueba vamos a partir del enunciado del teorema para de ahí explicar que pasos se van tomando para llegar al final de la prueba. El código entero en Lean puede leerse en el Anexo A.

```
theorem wop_aoc : well_ordered_principle γ → axiom_of_choice X
```

Para empezar vamos a introducir como hipótesis el principio de buena ordenación y el hecho de tener una familia de conjuntos no vacíos. Con esto se nos genera la meta de que existe una función de elección. Utilizamos la táctica **rw** para poder reescribir el principio de buena ordenación y así extraer como hipótesis la relación R y el hecho de que es un buen orden.

Ahora vamos a utilizar **refine'** para poder encontrar la f que cumple la propiedad buscada. En este caso crea las dos metas que son necesarias como son:

- f : I → ⋃ i, X i
- ∀ (i : I) (f i).1 ∈ X i

Para la primera meta se extrae como hipótesis que tenemos (i : I), ahora debemos utilizar **choose** sobre la hipótesis del principio de buena ordenación, que crea la meta de que existe un elemento

mínimo en (X i) que pertenece a la unión de todos los (X i). Con esto podemos reescribir la expresión con cuantificadores existenciales utilizando **rw[mem_iUnion]**, para la cual utilizamos la variable i. Entonces solo queda utilizar la propiedad **choose_spec** para determinar que el mínimo de (X i) está en (X i). Es importante destacar que, en ese caso, **choose_spec** se traduce en **choose** (_ : ∃ n ∈ X i, ∀ m ∈ X i, R n m) ∈ X i ∧ ∀ m ∈ X i, R (**choose** (_ : ∃ n ∈ X i, ∀ m ∈ X i, R n m)) m y para esta prueba únicamente necesitamos el campo **left** de esta conjunción.

Para la segunda, como hemos definido correctamente la primera parte, basta con probar que el elemento devuelto por la función f está en (X i). En efecto, necesitamos ver que este elemento escogido en la unión está en (X i), por lo tanto como f : I → ⋃ i, X i únicamente tenemos que elegir la propiedad de que un elemento mínimo pertenece al propio conjunto, de nuevo, se prueba con el campo **left** de **choose_spec**.

Con esto no quedan mas metas por probar con lo que queda demostrado que el principio de buena ordenación implica el axioma de elección.

6.3.2. El Lema de Zorn implica el Principio de Buena Ordenación

Para esta demostración se va a necesitar definir unos conceptos previos. El primero de ellos es una estructura que se define como **WellOrderedSet**, este va a tener 4 campos como son:

- Un conjunto de tipo Set γ.
- Una relación en dicho conjunto.
- La hipótesis de que el conjunto con esta relación conforman un buen orden.
- La hipótesis de que los elementos de fuera del conjunto no están relacionados.

```
@[ext]structure WellOrderedSet (γ : Type _) :=
(s : Set γ)
(r : γ  → γ  → Prop)
(h_well : is_well_ordered s (fun x y => r x.val y.val))
(h_triv : ∀ (x y : γ) (_ : x ∉ s ∨ y ∉ s), ¬ r x y)
```

Observación 6.3.1. La sintaxis **@[ext]** nos permite utilizar que dos elementos x, y de **WellOrderedSet** son iguales si lo son sus conjuntos y relaciones asociados.

Ahora queremos ordenar los elementos de este tipo que acabamos de definir. Para ello definimos una instancia de **PartialOrden** en el que vamos completando los diferentes campos que se solicitan para poder ordenar los elementos. Una instancia de esta clase requiere de definir varios campos como son los siguientes:

```
class PartialOrder(α : Type u) extends Preorder : Type u
le : α → α → Prop
lt : α → α → Prop
le_refl : ∀ (a : α), a ≤ a
le_trans : ∀ (a b c : α), a ≤ b → b ≤ c → a ≤ c
lt_iff_le_not_le : ∀ (a b : α), a < b ↔ a ≤ b ∧ ¬b ≤ a
le_antisymm : ∀ (a b : α), a ≤ b → b ≤ a → a = b
```

En el Anexo A podemos ver para el caso de **WellOrderedSet** como definimos y probamos cada propiedad de la clase. En un primer momento se dejan como **sorry**, asumiendo que las pruebas son correctas, para poder comprobar que los teoremas importantes se pueden probar. Una vez demostrados completamos las instancia haciendo los cambios necesarios para llegar a completar todas las metas. Aunque no vayamos a explicar las pruebas de las propiedades, vamos a mostrar de que manera ordenamos los elementos de **WellOrderedSet**.

```
le := fun x y => x.s ⊆ y.s ∧ ∀ (a b), x.r a b → y.r a b
lt := fun x y => x.s ⊊ y.s ∧ ∀ (a b), x.r a b → y.r a b
```

Entonces si tenemos dos elementos de **WellOrderedSet** uno es menor que otro si el conjunto del primero está incluido en el conjunto del segundo y si la relación del primero relaciona dos elementos también lo hace la del segundo. Es exactamente la forma de ordenar que hemos definido previamente para los pares (A, R).

Con esto pasamos a enunciar y demostrar un lema en el que vamos a ver que un conjunto F es inductivo, **F_is_inductive**. Esta prueba tiene como base definir el conjunto que hemos visto en el Capítulo 6. Para ello dada una cadena de **F : Set (WellOrderedSet γ)** vamos a definir **A** como la unión de los conjuntos de los elementos de la cadena y como **R** la relación de que existe un elemento en la cadena que relaciona dos elementos cualesquiera del conjunto. Con **A**, **R** y las correspondientes hipótesis **h_well** y **h_triv** podemos definir un elemento de **WellOrderedSet** al que llamaremos **ub**, siendo esta la cota superior de la cadena. Veamos como definimos en Lean el conjunto y relación de **ub**.

```
let A := ⋃ x ∈ c, x.s
set R := fun a b => ∃ cᵢ, cᵢ ∈ c ∧ cᵢ.r a b with hR
```

Una vez definido todo lo necesario utilizamos la táctica **use ub** para avanzar a la meta ∀ x ∈ c, x ≤ ub. Ahora dado un elemento de la cadena y ∈ c vamos a probar que está acotado superiormente por **ub**. Utilizando la definición que hemos dado para **le** necesitamos probar ambos lados de la conjunción.

- El lado izquierdo requiere probar que y.s $\subseteq \bigcup_{x \in c}$, x.s. Para probarlo utilizamos el teorema **subset_biUnion_of_mem** que dada la hipótesis de que y $\in$ c entonces y.s $\subseteq \bigcup_{y \in c}$, y.s que mediante **exact** unifica terminando la prueba.

- El lado de la derecha tiene como hipótesis dos elementos y el hecho de que están relacionados en y, es decir, y.r a b. Ahora queremos probar que $\exists$ $c_i \in$ c, c_i.r a b. Como tenemos la hipótesis de que y $\in$ c mediante **use** y, unificamos la meta.

Ahora que tenemos todo lo necesario pasamos a la prueba del teorema que buscamos, es decir, que el Lema de Zorn implica el principio de buena ordenación. Para ello vamos a tomar el elemento maximal (**m**) y vamos a probar que el conjunto asociado a él es todo el espacio para luego utilizar la relación asociada de dicho elemento para terminar de ver que **m** está bien ordenado.

Empezamos utilizando la táctica **obtain** ⟨m, hm⟩ := hZ para obtener el elemento maximal (**m**) y la hipótesis de que lo es (**hm**), extraídas del Lema de Zorn (**hZ**).

Para continuar con la prueba necesitamos probamos que el conjunto m.s = univ γ como resultado intermedio utilizando **have**. Como la meta actual es $\exists$ R, **is_well_ordered** γ R utilizamos m.r como relación para llegar a que la meta es **is_well_ordered** γ m.r. Por la propiedad **h_well** de **m** si pudiésemos aplicar el teorema que asegura que univ $\gamma \simeq \gamma$ entonces la meta que tenemos sería equivalente a **is_well_ordered** m.s m.r y por **h_well** unificaría.

Observación 6.3.2. A lo largo del código en proporcionado se pueden observar como hay ciertas sentencias **sorry**. Estas indican que hay partes de formalización en las que no se ha encontrado la sintaxis adecuada para terminar las pruebas, no obstante, la lógica requerida es la que hemos ido exponiendo a lo largo del capítulo. Por lo que, aún siguiendo una forma de trabajo correcta requiere de experiencia para terminar pruebas con matices determinados. Un ejemplo particular es el conseguir unificar el último paso de este teorema, donde sabemos que (↑ **Set.univ** γ) $\simeq \gamma$. En el caso de saber qué aplicar exactamente la prueba estaría hecha pero se puede ver que el resultado es correcto pues ambos tipos son equivalentes mediante el teorema **Equiv.Set.univ** que vemos en el código de debajo. El problema reside en que intentando utilizar teoremas de equivalencia (y no de igualdad) no ha sido posible llegar a unificar la meta de modo que sea resuelta correctamente.

```
def Equiv.Set.univ(α : Type u_1) : ↑Set.univ ≃ α
```

6.3.3. El Axioma de Elección implica el Lema de Zorn

Como comentamos al principio de este capítulo, únicamente vamos a abordar las pruebas de dos de las tres implicaciones. No obstante, vamos a ver el enunciado del teorema y del lema clave para dar luz al trabajo futuro que consistiría en formalizar este resultado. El teorema sigue la línea de los dos anteriores utilizando las definiciones ya vistas:

```
theorem aoc_implies_zorn : axiom_of_choice X → zorn F (F_is_inductive F)
```

Para continuar tenemos que definir una familia de conjuntos indexados A : I → Set γ y una función que mapea dichos conjuntos que denominamos φ : Set γ → Set γ. Ahora podemos definir el lema clave:

```
lemma key_lemma [PartialOrder γ]
(h₁ : (∀ (i : I), Nonempty (A i)) → is_chain γ (A i) → ∃ (j : I), A j = ⋃ i, A i)
(h₂ : ∀ (i : I), (A i) ⊆ φ (A i))
(h₃ : ∀(i : I), ∅ = (φ (A i) \ (A i)) ∨ ∃! (a : γ), a ∈ (φ (A i) \ (A i))) :
∃ (i : I), φ (A i) = (A i)
```

En primer lugar es necesario que γ sea un **PartialOrder** para que **is_chain** unifique. Ahora expliquemos las tres hipótesis:

- Para toda cadena no vacía de la familia A, existe otro elemento de la familia que es la unión de la cadena.
- Todo elemento de la familia está incluido en su imagen por φ.
- El conjunto φ (A i) es un *singleton*, es decir, o es el conjunto vacío o tiene un único elemento.

En caso de cumplirse entonces existe un elemento en la familia de modo que su imagen por φ coincide con el propio conjunto. Ahora sería necesario buscar la forma de definir una Tribu y un Cronopio en Lean y hacer las modificaciones necesarias para hacer la formalización lo más sencilla posible. Pero esto se queda fuera del objetivo de estudio.

Capítulo 7

Conclusiones y trabajo futuro

A lo largo de este documento hemos ido paso a paso estudiando como utilizar Lean 4 con diferentes ejemplos sencillos para terminar con un ejemplo real de como se utilizaría. Repasemos los objetivos que se proponían en un inicio justificando como se han conseguido:

- **Entender el fundamento teórico de Lean 4 y su funcionamiento mediante resultados sencillos.**

  Este objetivo ha sido cubierto en el capítulo 2 y en el capítulo 3, en los cuales hemos visto la teoría de tipos dependientes que subyace a Lean 4 así como el isomorfismo de Curry-Howards. Además hemos visto maneras básicas de crear definiciones y teoremas, así como aspectos lógicos básicos para manipular los enunciados y la interacción con la interfaz de Lean 4 para probar las metas.

- **Conocer las tácticas más conocidas de lean 4 así como sus usos.**

  También se ha logrado este objetivo a lo largo del capítulo 4 donde se ha ahondado en lo relativo a las diferentes tácticas más comunes y las utilizadas posteriormente para la prueba que se pretendía.

- **Entender un resultado conocido y su demostración clásica para tener un enfoque de como formalizarlo.**

  A lo largo del capítulo 5 se van definiendo conceptos como el Pincipio de buena ordenación, el Lema de Zorn y el Axioma de elección así como la importancia y usos de estas definiciones en otros resultados. Una vez hecho esta primera aproximación se demuestra la equivalencia entre estas tres definiciones.

- **Formalizar parte del resultado y explicar la lógica subyacente**

  Hemos conseguido formalizar que el Principio de buena ordenación implica el Axioma de elección, crear una estructura de conjuntos con buenos órdenes en ellos y crear una instancia

de **PartialOrder** de esta estructura y llegar a formalizar, salvo equivalencia y resultados intermedios, que el Lema de Zorn implica el principio de buena ordenación.

En resumen, podemos concluir que los objetivos han sido cumplidos con éxito, no obstante la formalización es un proceso largo y en el que se requiere que todos los tipos unifiquen de manera exacta. Por lo tanto, aún siendo la manera más fiable para verificar una demostración, requiere de mucho trabajo y tiempo para familiarizarse con la sintaxis del lenguaje. Esto nos deja posibilidad para un trabajo futuro muy prometedor con todo lo que ya se ha conseguido:

- **Revisar el trabajo hecho para compactar el código**

  Es cierto que todo el código que ya está hecho formaliza correctamente los resultados. Pero hay estructuras en Lean 4 pensadas para ahorrar líneas de código que podrían ser equivalentes a las utilizadas. Para ello sería cuestión de leer la documentación de teoremas que ya están en Mathlib 4 y refinar así el código.

- **Terminar las pruebas que restan y completar los *sorry***

  La última implicación y algunos ***sorry*** previos tendrían que ser resueltos para formalizar del todo el resultado. Se requerirá del uso de estructuras más complejas y de tácticas diferentes para que, al igual que los anteriores teoremas, quede completado.

Apéndice A

Equivalencia en Lean

En todo este anexo tenemos el código que está subido al proyecto de github [7].

```
import Mathlib.Tactic
import Mathlib.Init.Order.Defs

variable (γ : Type*)

--Familia de conjuntos indexada por I.
--Sea (i : I) entonces X i : Set γ
variable {I : Type*}
variable (X : I → Set γ)

variable (S : Set γ )

--Abrimos los espacios de nombres para usar los alias cortos.
open Set
open Classical

--Todo lo relativo al principio de buena ordenacion, definir que una relacion es buen
--orden para luego enunciar el principio de buena ordenacion.
def is_well_ordered (R : γ  → γ → Prop) : Prop := ∀ (A : Set γ), Nonempty A → ∃ n ∈ A,
    ∀ m ∈ A, R n m
def well_ordered_principle : Prop := ∃ (R : γ  → γ → Prop), is_well_ordered γ R

--Por comodidad el tipo γ va a ser implicito a partir de aqui.
variable {γ}

--Definicion del axioma de eleccion usando la familia indexada de conjuntos antes definida.
def axiom_of_choice : Prop := (∀ (i : I), Nonempty (X i)) → ∃ (f : (I → ⋃ i, X i)), ∀
    (i : I), (f i).1 ∈ X i

--Definicion de resultados previos (ser una cadena y ser inductivo) para luego
--poder formalizar el Lema de Zorn.
def is_chain [PartialOrder γ] (c : Set γ) : Prop := ∀ (x y : γ), x ∈ c → y ∈ c → x ≤ y
    ∨ y ≤ x
def inductive_set [PartialOrder γ] (S : Set γ) : Prop := ∀ (c : Set γ), c ⊆ S → is_chain
    c → ∃ (ub : γ), ∀ (x : γ ), x ∈ c → x ≤ ub
def zorn [PartialOrder γ] (_ : inductive_set S ) : Prop := ∃ (m : γ), m ∈ S ∧ ∀ (x : γ),
    x ∈ S → x < m

theorem wop_aoc : well_ordered_principle γ → axiom_of_choice X := by
```

```
--Introducimos hipotesis
  rintro wop nempty

--Reescribimos el PBO y obtenemos la hipotesis de ser buen orden R.
  rw[well_ordered_principle] at wop
  obtain ⟨R,hR⟩ := wop
  rw[is_well_ordered] at hR

--Usamos refine' para dividir en submetas la meta actual
  refine' ⟨ fun i => _,_⟩

--Demostramos la primera meta eligiendo un elemento que cumpla la propiedad y con
--mem_iUnion podemos aplicar la propiedad particularizando en i.
  . use choose (hR (X i) (nempty i))
    rw [mem_iUnion]
    use i
    exact (choose_spec (hR (X i) (nempty i))).left
--Por el otro lado solo tenemos que aplicar la propiedad que se cumple para todo i.
  . dsimp
    intro i
    exact (choose_spec (hR (X i) (nempty i))).left

--Definimos los pares (s,r) de la literatura con las propiedades de que r es un buen orden
--de s y que los elementos externos de s no áestn relacionados.
@[ext]structure WellOrderedSet (γ : Type _) :=
(s : Set γ)
(r : γ  → γ  → Prop)
(h_well : is_well_ordered s (fun x y => r x.val y.val))
(h_triv : ∀ (x y : γ) (_ : x ∉ s ∨ y ∉ s), ¬ r x y)

--Conjunto que vamos a comprobar que es inductivo.
variable (F : Set (WellOrderedSet γ))

--Instancia de PartialOrden para poder ordenar los WellOrderedSet
instance : PartialOrder (WellOrderedSet γ) where
  le := fun x y => x.s ⊆ y.s ∧ ∀ (a b), x.r a b → y.r a b
  lt := fun x y => x.s ⊊ y.s ∧ ∀ (a b), x.r a b → y.r a b
  le_refl := by
    intro x
    simp only [imp_self, implies_true]
    apply And.intro
    . apply Set.Subset.refl
    trivial

  le_trans := by
    intro x y z hxy hyz
    dsimp only
    apply And.intro
    . apply Set.Subset.trans (And.left hxy) (And.left hyz)
    intro a b hxr
    apply And.right hyz a b
    apply And.right hxy a b
    exact hxr

  lt_iff_le_not_le := by
    intro x y
    apply Iff.intro
    . intro h
      apply And.intro
```

```
    . apply And.intro
      . simp at h
        exact And.left h.1
      intro a b hxr
      simp at h
      apply And.right h a b
      exact hxr
    intro h_le_yx
    have not_h_s := And.right (And.left h)
    have h_s := And.left h_le_yx
    contradiction
  intro h
  apply And.intro
  . rw[ssubset_def]
    apply And.intro
    . exact h.1.1
    rw[not_subset]
    simp only [not_and, not_forall, exists_prop] at h
    by_cases hyx : y.s ⊆ x.s
    . have h2 := h.2 hyx
      rcases h2 with ⟨x1,x2,h_y_r,h_x_not_r⟩
      by_contra nota
      push_neg at nota
      have h3 := y.h_triv
      specialize h3 x1 x2
      rw[←not_imp_not] at h3
      simp only [not_not, not_or] at h3
      specialize h3 h_y_r
      have x1x : x1 ∈ x.s := by
        apply nota
        exact h3.1
      have x2x : x2 ∈ x.s := by
        apply nota
        exact h3.2
      have h4 := x.h_triv
      specialize h4 x1 x2
      --rw[←not_imp_not] at h4
      --simp only [not_not, not_or] at h4
      have h5 : x1 ∈ x.s ∧ x2 ∈ x.s := by
        apply And.intro x1x x2x
      sorry
    rwa[not_subset] at hyx
  intro a b hxab
  apply h.1.2
  exact hxab


le_antisymm := by
  intro x y le_xy le_yx
  have s_eq : x.s = y.s := Set.Subset.antisymm le_xy.1 le_yx.1
  have r_eq : ∀ a b, (x.r a b ↔ y.r a b) := by
    intro a b
    apply Iff.intro
    . intro h
      exact le_xy.2 a b h
    intro h
    exact le_yx.2 a b h
  rcases x with ⟨x.s,x.r,x.hwell,x.htriv⟩
  rcases y with ⟨y.s,y.r,y.hwell,y.htriv⟩
```

```
    simp only [WellOrderedSet.mk.injEq]
    apply And.intro
    . exact s_eq
    ext a b
    apply r_eq


--Lema necesario para porbar que el conjunto F es inductivo.
lemma F_is_inductive : inductive_set F := by
  intro c h1 h2
  let A := ⋃ x ∈ c, x.s
  set R := fun a b => ∃ c_i, c_i ∈ c ∧ c_i.r a b with hR

  let ub : WellOrderedSet γ :={
    s:=A
    r:=R
    h_well:=by
      rw[is_well_ordered]
      intro A_i hA_i_ne
      obtain ⟨n,hn⟩ := hA_i_ne
      use n
      apply And.intro
      . apply hn
      intro m hm
      rw[hR]
      dsimp only
      have h_n : ∃ x ∈ c, n.1 ∈ x.s := by
        simp only at n
        obtain ⟨x1, hx1⟩ := n
        rw[mem_iUnion] at hx1
        obtain ⟨x, hx⟩ := hx1
        use x
        rw[mem_iUnion] at hx
        obtain ⟨xc,x1s⟩ := hx
        apply And.intro
        . exact xc
        exact x1s
      have h_m : ∃ x ∈ c, m.1 ∈ x.s := by
        simp only at m
        obtain ⟨x1, hx1⟩ := m
        rw[mem_iUnion] at hx1
        obtain ⟨x, hx⟩ := hx1
        use x
        rw[mem_iUnion] at hx
        obtain ⟨xc,x1s⟩ := hx
        apply And.intro
        . exact xc
        exact x1s
      obtain ⟨c_n, hc_n⟩ := h_n
      obtain ⟨c_m, hc_m⟩ := h_m
      have h_chain := h2 c_n c_m hc_n.left hc_m.left
      rcases h_chain with hleft | hright
      --Busco encontrar que estan relacionados
      . use c_n
        apply And.intro hc_n.1
        sorry
      use c_m
      sorry
    h_triv := by
```

```
      intro a1 a2 h
      simp only [not_exists, not_and]
      intro c_i hc_i
      have h_triv := c_i.h_triv
      apply h_triv
      rw[mem_iUnion,mem_iUnion] at h
      simp only [mem_iUnion, exists_prop, not_exists, not_and] at h
      rcases h with hleft | hright
      . apply Or.intro_left
        apply hleft
        exact hc_i
      apply Or.intro_right
      apply hright
      exact hc_i
  }
  use ub
  intro y hy
  apply And.intro
  . simp only
    exact subset_biUnion_of_mem hy
  intro a b hyr
  simp only
  use y

--Demostracion de la implicacion LZ → PBO utilizando que F es inductivo
theorem zorn_implies_wop : zorn F (F_is_inductive F) → well_ordered_principle γ := by

--Obtenemos las hipotesis del Lema de Zorn, un m maximal y hm la propiedad de ser maximal.
  intro hZ
  rw[zorn] at hZ
  rw[well_ordered_principle]
  obtain ⟨m, hm⟩ := hZ

--Probamos que m.s es todo el universo.
  have : m.s = Set.univ := by
    apply eq_univ_of_forall
    intro x
    by_contra hx
    have h := hm.2
    let M : WellOrderedSet γ := {
      --ñAado x al conjunto m.s
      s:= {x} ∪ m.s
      --x es el elemento ámas grande del conjunto
      r:= fun a b => if a = x then false else if b = x then true else m.r a b


      h_well := by
        rw[is_well_ordered]
        intro A hA_ne
        have h3 := m.3
        rw[is_well_ordered] at h3
        sorry


      h_triv := by
        intro a b h1
        simp
        intro ha
        have h2 := m.4
```

```
        specialize h2 a b
        apply And.intro
        . sorry
        apply h2
        apply Or.intro_left
        apply Or.elim h1
        . sorry
        sorry

    }
    have hmM : m ≤ M := by
      apply And.intro
      . simp only [singleton_union, subset_insert]
      intro a b mr
      simp only [Bool.ite_eq_true_distrib, ite_eq_left_iff, decide_eq_true_eq,
    if_false_left_eq_and]
      apply And.intro
      . sorry
      sorry
    have hMF : M ∈ F := sorry
    have hMm : M < m := by
      apply h
      exact hMF
    simp only[LE.le] at hmM
    simp only[LT.lt] at hMm
    have h1 := hMm.1
    have h2 : x ∈ m.s := by
      have : {x} ∪ m.s ⊆ m.s := by
        exact subset_of_ssubset h1
      rw[union_subset_iff] at this
      simp only [singleton_subset_iff] at this
      exact this.1
    contradiction

--Utilizamos la relacion de m (m.r) que sera la que cumpla lo que pide la meta.
  use m.r

--Sabemos que m es un buen orden por construccion
  have h3 := m.3

--Aplicamos que m.s = univ γ
  rw[this] at h3

--Tenemos que ↑univ es equivalente a γ
  have equiv := Equiv.Set.univ γ

--Ver si puedo usar la equivalencia de alguna manera son Equiv.Set.univ
  sorry

theorem aoc_implies_zorn : axiom_of_choice X → zorn F (F_is_inductive F) := by
  sorry

variable (A : I → Set γ )
variable (φ : Set γ  → Set γ )

lemma key_lemma [PartialOrder γ] (h₁ : (∀ (i : I), Nonempty (A i)) → is_chain (A i) → ∃
    (j : I), A j = ⋃ i, A i) (h₂ : ∀ (i : I), (A i) ⊆ φ (A i))
(h₃ : ∀(i : I), ∅ = (φ (A i) \ (A i)) ∨ ∃! (a : γ), a ∈ (φ (A i) \ (A i))) : ∃ (i : I), φ
    (A i) = (A i) := by
```

```
sorry
```

Bibliografía

[1] J. Avigad and P. Massot. *Mathematics in Lean.*

[2] J. Avigad, P. Massot, S. Kong, S. Ullrich, and L. Community. *Theorem Proving in Lean 4.*

[3] P. Bonilla Nadal. Dependent typing through closed cartesian cathegories. Master's thesis, Universidad Autónoma de Madrid, 2021.

[4] A. Cordoba. *La saga de los numeros.* Editorial Crítica, 2006.

[5] D. Eisenbud. *Commutative Algebra with a View Toward Algebraic Geometry.* Springer, 1995.

[6] J. F. Fernando and J. M. Gamboa. *Estructuras Algebraicas : Divisibilidad en Anillos Conmutativos.* Editorial Sanz y Torres, 2017.

[7] S. Mourenza Rivero. Equivalence of well-ordering principle, zorn's lemma, and axiom of choice. https://github.com/Santimr19/AC_iff_ZL_iff_WOP, 2024.

[8] G. Smolka and J. Schwinghammer. *Lecture Notes for Semantics.* 2008.

[9] K. Stromberg. The banach-tarski paradox. *Amer. Math. Monthly*, 86:151–156, 1979.

[10] D. van Dalen. *Logic and Structure.* Springer, 2008.

[11] G. Vitali. Sui gruppi di puunti e sulle funzioni di dariabli reali. *Atti Accad. Sci. Torino*, 43:75–92, 1908.