Trabajo de Fin de Grado

Formalización de las matemáticas con Lean.
Un caso de estudio: Geometría euclídea plana.

Adrián Lattes Grassi

Septiembre de 2023

Facultad de Ciencias Matemáticas.
Trabajo dirigido por Jorge Carmona Ruber.

Resumen

Este trabajo explora el uso del asistente de demostración *Lean*, un lenguaje de programación que implementa una teoría de tipos útil para verificar formalmente demostraciones matemáticas, para formalizar enunciados y resultados de la axiomática de Hilbert de la geometría euclídea plana. Esta teoría servirá de guía para introducir el uso del asistente y exponer cómo puede ser utilizado para construir relaciones de equivalencia, modelos de una teoría y demostrar la independencia entre axiomas.

Abstract

Resumen traducido al inglés.

Introducción

Los asistentes de demostración nos permiten formalizar definiciones, enunciados de proposiciones y teoremas, demostraciones, y verficar estas definiciones. Formalizar matemáticas consiste en digitalizar enunciados y resultados escribiéndolos en un lenguaje de programación que garantiza, mediante una correspondencia entre una teoríá de tipos y la lógica, la validez de cada paso.

Algunos beneficios de formalizar enunciados y resultados matemáticos mediante un asistente de demostración son:

1. Matemáticas y geometría formales.

La formalización matemática es el proceso de representar enunciados y demostraciones matemáticas utilizando un lenguaje formal y un conjunto de reglas lógicas bien definidas. En este contexto, se establece un conjunto de proposiciones fundamentales, conocidas como axiomas, que se aceptan sin requerir justificación. Estos axiomas se manipulan y transforman mediante la aplicación sistemática de las reglas lógicas para derivar nuevas proposiciones matemáticas. El proceso de aplicar secuencialmente estas reglas lógicas para obtener una conclusión a partir de proposiciones más básicas es conocido como demostración.

En la historia de las matemáticas los *Elementos* de *Euclides*, tratado matemático compuesto de trece libros escrito en el siglo III a.C., son el ejemplo más antiguo de proyecto de formalización matemática. En la obra se trata rigurosamente una extensa variedad de temas, como la geometría plana y espacial o la teoría de números. Euclides es el primer autor en presentar los conocimientos matemáticos siguiendo un método formal. En los tratados se presentan los argumentos a partir de una serie de postulados, definiciones y nociones comunes a partir de los cuales se demuestran proposiciones y teoremas mediante razonamientos deductivos.

La obra de Euclides ha tenido una profunda influencia en las matemáticas, la lógica y la filosofía. Los *Elementos* se mantuvieron como la principal referencia en geometría durante casi dos milenios. Las pruebas rigurosas, apoyadas en el razonamiento lógico y la estructura del tratado establecieron un estándar para la argumentación y la exposición matemática que todavía se conserva en la actualidad.

A lo largo de la historia, se ha evidenciado que los *Elementos* de *Euclides* no se ciñen estrictamente al método axiomático. En el tratado se encuentran razonamientos sustentados en intuiciones geométricas y construcciones con regla y compás, en lugar de en deducciones estrictamente lógicas derivadas de los postulados.

Durante siglos, matemáticos y filósofos han examinado y escrutado la obra de Euclides, identificando errores y omisiones y planteando cuestiones sobre la relación entre los postulados. El quinto postulado de Euclides, también conocido como postulado de las paralelas, ha sido sometido a un análisis riguroso. Este postulado afirma que, dada una línea recta y un punto fuera de ella, existe únicamente una línea recta paralela a la línea dada que pasa por el punto en cuestión. Por siglos, numerosos matemáticos, intuyendo la innecesariedad del postulado, intentaron demostrarlo a partir de los otros cuatro, pero sin éxito. El problema fue resuelto en el siglo XIX con la concepción de nuevas geometrías, como la hiperbólica y la elíptica, en las cuales el postulado de las paralelas no se verifica. Quedó así demostrada la independencia del quinto postulado respecto a los primeros cuatro, y por tanto su necesidad para la construcción de la geometría euclídea.

Durante el siglo XIX se dieron avances trascendentales en el desarrollo de las matemáticas y la lógica formales. Ejemplos notables son la formulación del álgebra de Boole, la lógica de predicados propuesta por *Gottlob Frege* o el desarrollo de la *aritmética de Peano*. En este contexto el matemático alemán *David Hilbert* publicó su obra *Grundlagen der Geometrie* (Fundamentos de Geometría), en la cual se lleva a cabo una revisión exhaustiva de los *Elementos*, planteando una nueva axiomatización para formalizar correctamente los resultados de la geometría euclídea, eliminando por completo el recurso a la intuición y razonamientos geométricos en la presentación de los resultados.

Explicar que este es el punto de partida en el que enmarcar este trabajo. A partir de aqui se va a explicar qué aporta la formalización matemática asistida por computadores y en que sentido es un paso más.

2. Formalización asistida por computadores

Introducción a la formalización en Lean. Demostración asistida por computadores. ¿Pa que?

- Comprobación mecanizada de demostraciones.
- Digitalizar resultados y crear base de datos. FormalAbstracts
- Investigación en técnicas de demostración automática

3. Introducción a Lean

3.1. Teoría de tipos informal

Lo primero que se necesita en matemáticas para poder formalizar afirmaciones es un lenguaje formal con el cual expresarlas. Normalmente se utiliza la lógica de primer orden, en el contexto de la teoría de conjuntos. Lean, sin embargo, utiliza un sistema deductivo diferente, el de la teoría de tipos.

En lógica de primer se tiene una aserción fundamental, *que una proposición dada tenga una prueba*. Es decir, cada proposición P da lugar a la aserción correspondiente *P tiene una prueba*. Mediante ciertas reglas de transformación, y a veces una serie de axiomas, se pueden construir nuevas pruebas. Por ejemplo, dada la regla de inferencia *de A se deduce A o B* y la aserción *A tiene una prueba*, se obtiene la aserción *A o B tiene una prueba*.

En teoría de tipos la aserción fundamental es *que un término tenga un tipo*. Dados un término a y un tipo A, si *a tiene tipo A* escribimos $a : A$. Esta es misma notación es la utilizada por Lean, en el que por ejemplo podemos expresar la aserción *3 es un número natural* con el código `3 : ℕ`.

Es importante no confundir esta notación con la de una relación interna a nuestro lenguaje. Mientras que en teoría de conjuntos utilizamos la relación de pertenencia $\in$ para expresar que un elemento primitivo (un conjunto) está contenido en otro, en teoría de tipos no podemos considerar los términos o los tipos por separado. La noción fundamental es la pertenencia de tipos y cada término tiene que estar siempre acompañado por su tipo. En teoría de tipos además existen otras aserciones, como la de igualdad entre términos de un mismo tipo. Dados $a : A$ y $b : A$, se tiene la aserción *a y b son dos términos de tipo A iguales por definición*, y escribimos $a \equiv b : A$.

La teoría de tipos también puede utilizarse para expresar afirmaciones y demostraciones matemáticas. Las afirmaciones se codifican mediante los tipos y las demostraciones mediante construcciones de términos de un tipo dado. Es decir, se puede interpretar la aserción $a : A$ como *a es una demostración de A*. Esta interpretación da lugar a una analogía entre la lógica proposicional y la teoría de tipos, llamada correspondencia de Curry-Howard. Sin entrar en detalles, a cada proposición lógica se le puede asignar un tipo, y a cada demostración de un enunciado un término del tipo correspondiente al enunciado.

Referencia: HTT

Existen distintas elecciones de reglas de transformación que considerar en teoría de tipos, que dan lugares a distintas versiones de la teoría de tipos. Lean implementa una una versión de la teoría de tipos dependiente conocida como el *Calculus of Constructions*.

La base de muchas teorías de tipos es el lambda cálculo, un modelo universal de computación introducido por *Alonzo Church* en los años treinta. Sin entrar en su formalización, en el lambda cálculo se consideran dos operaciones fundamentales para tratar con funciones, la abstracción y la evaluación.

- **Abstracción**. Es el mecanismo de definición de funciones mediante la introducción de parámetros. Dado un término `x + 1 : ℕ`, mediante la sintaxis `λ x : ℕ, (x + 1 : ℕ)` se convierte la variable libre `x` en una variable ligada por la abstracción, a la que llamamos parámetro de la función.

Es importante recordar que cada término tiene que ir acompañado del tipo al que pertenece. En esta expresión estamos indicando que tanto el parámetro `x` como el resultado de la función, `x+1`, son de tipo ℕ. Es decir, tenemos una función que dado un número natural devuelve otro número natural. Esto también puede escribirse como `(λ x, x + 1) : ℕ → ℕ`.

Lean además incluye notación para definir funciones que devuelven otras funciones, lo cual es muy útil para tratar funciones que reciben más de un parámetro (**Referencia:** Ver currificacion). Las siguientes líneas de código definen expresiones equivalentes

```
λ a : α, λ b : β, a
λ (a : α) (b : β), a
```

que representan el mismo término, de tipo `α → β → α`.

- **Evaluación**. Consiste en aplicar funciones, pasándoles los valores de los argumentos que evaluar. Por ejemplo la expresión `(λ x : ℕ, (x + 1) : ℕ) (1 : ℕ)` indica que estamos evaluando la función `(λ x, x + 1) : ℕ → ℕ` con el parámetro `x` sustituido por el término `1 : ℕ` (para que la sustitución pueda realizarse los tipos deben coincidir). Mediante un proceso de reducción se obtiene el término `2 : ℕ`.

Referencia: Theorem proving in Lean [AvD]

Como se ve en los ejemplos, el código fuente de Lean se pueden incluir caracteres unicode, como λ, → o ℕ. Esta característica del lenguaje es muy útil a la hora de escribir código lo más cercano posible a las notaciones a las que estamos acostumbrados a usar en matemáticas. La inclusión de estos caracteres está facilitada en el entorno de desarrollo, escribiendo comandos que empiecen por \ estos se reemplazarán por el caracter correspondiente. Por ejemplo al escribir `\to` este comando se reemplazará automáticamente por el caracter →, `\lambda` por λ y `\N` por ℕ.

Pendiente: Incluir en algún lugar explicación sobre entorno de desarrollo (plugin vscode) y sus características (¿Apéndice?).

Que cada término sea siempre considerado junto a su tipo no significa que sea siempre necesario explicitar dicho tipo. Lean tiene un mecanismo llamado *inferencia de tipos* que le permite deducir automáticamente el tipo de un término cuando no ha sido explicitado pero el contexto aporta información suficiente. Por ejemplo, cuando definimos la función `λ x : ℕ, (x + 1 : ℕ)` no es necesario incluir la segunda anotación de tipo. Dada la expresión `x + 1` y sabiendo por contexto que `x : ℕ` el sistema de inferencia de tipos deduce que la suma de dos números naturales también es un número natural, por lo que se infiere el tipo ℕ.

3.2. Introducción de términos

En lean existen distintas formas de introducir nuevos términos en el entorno actual.

Constantes

Mediante los comandos `constant` y `constants` se pueden introducir términos en el entorno, postulando su existencia. Este comando equivale por tanto a considerar nuevos axiomas sobre la existencia de los términos que introduce.

```
constant a : ℕ
constants (b : ℤ) (c : ℂ)
```

Definiciones

Si no queremos introducir nuevos axiomas podemos definir nuevos términos mediante el comando `def`. Como estamos definiendo un símbolo, es necesario proporcionar el tipo y término que queremos asignar al símbolo:

```
def succ : ℕ → ℕ := λ n, n + 1
def succ' (n : ℕ) : ℕ := n + 1 -- Otra forma de definir succ
```

Para introducir parámetros de funciones se puede omitir la notación λ, incluyendo las variables parametrizadas antes de los dos puntos que anotan el tipo de la definición.

Esta notación de introducción de parámetros es muy útil y simple, pero puede resultar demasiado explícita. Veamos por ejemplo cómo se puede definir la función identidad, que dado un término de un tipo devuelve el mismo término. Si queremos que la identidad definida sea sea general y se le puedan aplicar términos de cualquier tipo necesitaremos introducir el tipo del término como un parámetro adicional.

```
def id1 (α : Type*) (e : α) := e
```

El problema de esta definición es que cada vez que se quiera utilizar será necesario proporcionarle como primer argumento el tipo del término que se le quiere pasar, por ejemplo `id1 ℕ 0`. Pero la función identidad que queremos considerar recibe un solo argumento. Como a cada término acompaña siempre su tipo, dado el término `e : α`, el sistema de inferencia de tipos de Lean es capaz de deducir automáticamente el tipo α. Solo falta indicar en la definición cuál es el parámetro cuya identificación queremos delegar al sistema de inferencia de tipos.

```
def id2 {α : Type*} (e : α) := e
```

Así los parámetros indicados entre llaves, llamados *parámetros implícitos*, serán deducidos automaticamente.

3.3. Proposiciones

En Lean se tiene el tipo `Prop` para expresar las proposiciones mediante la analogía de *proposiciones como tipos* dada por la correspondencia de Curry-Howard.

La correspondencia de Curry-Howard afirma que estas funciones de la teoría de tipos se comportan de la misma forma que la implicación en lógica. Por tanto en Lean utilizaremos el símbolo `→` para referirnos tanto a funciones como a implicaciones dentro de una proposición. Dadas dos proposiciones `p q : Prop` podemos construir la proposición `p → q : Prop`, que se interpreta como `p` *implica* `q`.

Pendiente: Explicar cómo se expresan las proposiciones. or, and, neg, forall, exists, neq, etc

3.4. Demostraciones

Pendiente: Explicar qué es una demostración en teoría de tipos (term mode). Explicar qué son las tácticas y cómo funciona el modo táctico.

Pendiente: Incluir anexo hablando sobre el entorno de desarrollo y el modo táctico, con captura de pantalla.

3.5. Mathlib

Pendiente: Contar alguna cosa básica. Explicar cómo utilizar la web para encontrar cosas.

4. Formalizando la geometría de Hilbert en Lean

En esta sección se presentan algunos axiomas y resultados elementales de la axiomática de Hilbert, comparando los enunciados y demostraciones expresados de forma natural con sus correspondientes formalizaciones en Lean.

En 1899 Hilbert empezó a desarrollar su propuesta de nueva fundamentación de la geometría euclídea, mediante una serie de apuntes de conferencias que más tarde se convertirían en el tratado *Grundlagen der Geometrie* (Fundamentos de Geometría). Este trabajo hace énfasis en los problemas de clasificación de nociones primitivas, grupos de axiomas, interdependencias entre las distintas partes de la teoría y minimalidad de los axiomas considerados.

Esta nueva teoría geométrica parte de postular ciertas nociones primitivas y una serie de axiomas que establecen cómo se relacionan estas nociones. En este trabajo se ha seguido una versión modernizada de los axiomas y los resultadas, basada en las presentaciones de los libros de *Hartshorne* [Har00] y *Greenberg* [Gre93], en las que se considera el caso restringido de la geometría plana.

Consideraremos por tanto cinco nociones primitivas: dos términos primitivos (*puntos* y *líneas*) y cuatro relaciones primitivas (*incidencia*, *orden*, *congruencia de segmentos* y *congruencia de ángulos*).

Se tratarán los axiomas y definiciones correspondientes a estas nociones primitivas, siguiendo la estructura del tratado, mencionando alguna cuestión sobre el axioma de las paralelas, pero sin entrar en cuestiones de continuidad. Además se incluirá alguna formalización de resultados demostrables con las nociones presentadas.

4.1. Geometría de incidencia

El primer grupo de axiomas establece propiedades de la relación de *incidencia*, una relación binaria entre *puntos* y *líneas*. Dado un punto A y una recta l escribiremos $A \sim l$ para denotar que A y l están relacionados mediante la relación de incidencia. Los tres *axiomas de incidencia* son los siguientes:

Axioma I1. *Para cada par de puntos distintos A y B existe una única recta que los contiene.*

Axioma I2. *Cada línea contiene al menos dos puntos distintos.*

Axioma I3. *Existen tres puntos no colineares. Es decir, existen A, B y C tales que $AB \neq BC$.*

Para formalizar estos axiomas en Lean podríamos utilizar el comando `constant` o `axiom`, pero existen otras construcciones que permiten explicitar mejor y tener más control sobre qué axiomas se están usando en cada momento. En lugar de enunciar un axioma y a partir de entonces darlo siempre por válido, definiremos un objeto (un nuevo tipo) en el que agruparemos nociones primitivas y axiomas sobre estas.

Se procederá de forma análoga a la definición usual de un *grupo*, en el se consideran un conjunto (en *Lean* trabajaremos con tipos) con una operación, un elemento distinguido y unos axiomas. Para definir la *geometría de incidencia* se toman dos tipos (uno para los puntos y otro para las líneas), una relación entre estos tipos (la incidencia) y los axiomas.

Para hacer esto *Lean* consta de dos construcciones muy similares, las *estructuras* y las *clases*, mediante las cuales se pueden agrupar tipos y proposiciones sobre estos tipos. **Pendiente:** Explicar brevemente diferencia entre estructuras y clases. Esta es la definición de clase mediante la que hemos digitalizado las nociones y conceptos de la *geometría de incidencia*:

```
17 class incidence_geometry (Point Line : Type*) :=
18   (lies_on : Point → Line → Prop)
19   (infix ` ~ ` : 50 := lies_on)
20   (I1 {A B : Point} (h : A ≠ B) : ∃! l : Line, A ~ l ∧ B ~ l)
21   (I2 (l : Line) : ∃ A B : Point, A ≠ B ∧ A ~ l ∧ B ~ l)
22   (I3 : ∃ A B C : Point, neq3 A B C ∧ ¬ ∃ l : Line,  A ~ l ∧ B ~ l ∧ C ~ l)
```

`src/incidence_geometry/basic.lean`

Como se puede intuir leyendo el código, los tipos `Point` y `Line` son parámetros de la clase. Las siguientes líneas explicitan términos que tienen que existir y tener el tipo especificado después de los dos puntos.

La relación de incidencia se ha formalizado como una función que dados un punto y una línea devuelve la proposición que determina si el punto está en la línea. En la línea 19 se introduce la notación mencionada anteriormente, mediante el comando `infix`. En los axiomas, siguiendo el estilo de la libería *mathlib*, se ha evitado el uso de cuantificadores universales, y en su lugar se han incluido los términos correspondientes como parámetros, lo que es equivalente en teoría de tipos pero más cómodo de leer.

Es interesante observar lo cercano que es el código en *Lean* a la forma en la que escribiríamos los axiomas utilizando los símbolos usuales de la lógica. No es necesario tener conocimientos de *Lean* para entender la mayoría de los enunciados (no se da el mismo caso con las demostraciones).

Pendiente: Explicar cómo se usan las clases. Parámetros con corchetes e instancias.

4.1.1. Definiciones

Las siguientes definiciones son útiles para tratar con puntos y líneas y continuar el desarrollo de la teoría.

En las demostraciones es útil tenér una forma de, dados dos puntos distintos, construir el término de la línea que pasa por ellos, aprovechando el axioma I1.

```
noncomputable def line
  {Point : Type*} (Line : Type*) [incidence_geometry Point Line]
  {A B : Point} (h : A ≠ B) :
  { l : Line // A ~ l ∧  B ~ l } :=
begin
```

```
35    let hAB := I1 h,
36    rw exists_unique at hAB,
37    let P := λ l : Line, A ~ l ∧  B ~ l,
38    have hlP : ∃ l : Line, P l, { tauto },
39    exact classical.indefinite_description P hlP,
40  end
```

src/incidence_geometry/basic.lean

El tipo de esta definición (línea 33) es un *tipo dependiente*: al tipo `Line` se le asocia la propiedad de que los puntos pertenezcan al término correspondiente. En el código fuente completo hemos desarrollado también una versión de esta función, `line_unique`, que también devuelve la propiedad de unicidad dada por el axioma I2.

Definición (Colinearidad)**.** *Decimos que tres puntos distintos son* ***colineares*** *si existe una línea que los contiene (todos los puntos inciden en la línea).*

```
def collinear {Point : Type*} (Line: Type*) [incidence_geometry Point Line]
  (A B C : Point) : Prop := ∃ l : Line, A ~ l ∧ B ~ l ∧ C ~ l
```

src/incidence_geometry/basic.lean

En esta definición el parámetro del tipo `Point : Type*` es implícito, puesto que se puede inferir a partir de los términos `A B C : Point`. `Line : Type*` sin embargo tiene que ser explícito ya que los demás parámetros no proporcionan información suficiente para inferirlo automáticamente.

Definición (Puntos comunes)**.** *Decimos que un punto es* ***común*** *a dos líneas si está en ambas líneas. Si dadas dos líneas existe un punto común, decimos que las líneas* ***tienen*** *un punto en común.*

```
def is_common_point
  {Point Line : Type*} [incidence_geometry Point Line]
  (A : Point) (l m : Line) :=
  A ~ l ∧ A ~ m
```

src/incidence_geometry/basic.lean

En este caso los parámetros `Point Line : Type*` son ambos implícitos porque los argumentos `A : Point` y `l m : Line` proporcionan la información necesaria para la inferencia automática.

```
def have_common_point
  (Point : Type*) {Line : Type*} [incidence_geometry Point Line]
  (l m : Line) :=
  ∃ A : Point, is_common_point A l m
```

src/incidence_geometry/basic.lean

4.1.2. Resultados elementales

Uno de los primeros resultados que se pueden demostrar, utilizando sólamente los axiomas de incidencia es el siguiente:

Proposición 1. *Dos líneas distintas pueden tener como mucho un punto en común.*

Demostración. Sean l y m dos líneas. Supongamos que ambas contienen los puntos A y B con $A \neq B$. Por el axioma I1, existe una única línea que pasa por A y B, por lo que l y m deben ser iguales. □

Esta demostración, que sigue la presentación del libro de *Hartshorne* [Har00], puede interpretarse como una demostración por reducción al absurdo sobre la condición de que las dos líneas sean iguales, o como una demostración por contraposición: si asumimos que no se cumple la conclusión (que las dos lineas no tengan más de un punto en común), entonces tampoco se cumple la premisa (que las dos líneas sean iguales).

Al implementar estas ideas en Lean nos damos cuenta de que hay bastantes detalles que necesitamos tener en cuenta.

```
21 lemma neq_lines_have_at_most_one_common_point
22   (Point : Type*) {Line : Type*} [ig : incidence_geometry Point Line] :
23   ∀ l m : Line, l ≠ m →
24     (∃! A : Point, is_common_point A l m)
25     ∨ (¬ have_common_point Point l m) :=
26 begin
27   intros l m,
28   contrapose,
29   push_neg,
30   rintro ⟨not_unique, hlm⟩,
31   rw exists_unique at not_unique,
32   push_neg at not_unique,
33   cases hlm with A hA,
34   rcases not_unique A hA with ⟨B, ⟨hB, hAB⟩⟩,
35   rw ne_comm at hAB,
36   exact unique_of_exists_unique (ig.I1 hAB) ⟨hA.1, hB.1⟩ ⟨hA.2, hB.2⟩
37 end
```

src/incidence_geometry/propositions.lean

Analicemos la demostración línea por línea:

L.26 El estado táctico inicial incluye los parámetros del lema. En este caso los tipos `Point` y `Line` e `ig`, la instancia de la clase `incidence_geometry`. Esta instancia representa el hecho de que los tipos `Point` y `Line` cumplen los axiomas de la geometría de incidencia.

La meta se corresponde con el enunciado del lema, es decir lo que queremos demostrar.

L.27 `intros l m`, La aplicación de la táctica `intros` introduce las hipótesis `l` y `m`. Es decir, saca el cuantificador universal de la meta e introduce las variables cuantificadas en el estado táctico, pasando a tener ahora dos nuevos términos `l : Line` y `m : Line`. La nueva meta es

```
l ≠ m → (∃! (A : Point), is_common_point A l m) ∨ ¬have_common_point Point l m
```

Esto equivale a decir en lenguaje natural "sean l y m dos líneas"

L.28 `contrapose`, La táctica `contrapose` permite realizar una demostración por contraposición. Es decir, si nuestra meta es de la forma `A → B`, la reemplaza por `¬B → ¬A`. En este caso la meta resultante es

```
⊢ ¬((∃! (A : Point), is_common_point A l m) ∨ ¬have_common_point Point l m) → ¬l ≠ m
```

L.29 `push_neg`, La táctica `push_neg` utiliza equivalencias lógicas para «empujar»las negaciones dentro de la fórmula. En este caso, al no haber especificado una hipótesis concreta, se aplica sobre la meta.

En la primera parte de la implicación se aplica una ley de De Morgan para introducir la negación dentro de una disjunción, convirtiéndola en una conjunción de negaciones. En la segunda, negar una desigualdad equivale a una igualdad. Por tanto la meta resultante es

```
⊢ (¬∃! (A : Point), is_common_point A l m) ∧ have_common_point Point l m → l = m
```

Es interesante notar que `push_neg` no consigue 'empujar' la negación todo lo que podríamos desear.

Esto es así porque no está reescribiendo las definiciones previas y de `∃!`. Esto lo tendremos que hacer manualmente, como se verá enseguida.

L.30 `rintro ⟨not_unique, hlm⟩`, La táctica `rintro` funciona como `intro`, en este caso aplicada para asumir la hipótesis de la implicación que queremos demostrar. La variante `rintro` nos permite entrar en definiciones recursivas, en este caso en la del operador `∧`, y mediante el uso de los paréntesis ⟨⟩ introducir los dos lados de la conjunción como hipótesis separadas. Por tanto después de aplicar esta táctica obtendremos dos hipótesis adicionales:

```
not_unique: ¬∃! (A : Point), is_common_point A l m
hlm: have_common_point Point l m
```

y la meta resultante es el segundo lado de la implicación, es decir `⊢ l = m`.

L.31 `rw exists_unique at not_unique,` La táctica `rw` (abreviación de `rewrite`) nos permite reescribir ocurrencias de fórmulas utilizando definiciones o lemas de la forma `A ↔ B`. Al escribir `at` indicamos dónde queremos realizar dicha reescritura, en este caso en la hipótesis `not_unique`.

En este caso utilizamos la definición de `∃!`, con lo que se modifica la hipótesis

```
not_unique : ¬∃ (x : Point),
is_common_point x l m ∧ ∀ (y : Point), is_common_point y l m → y = x
```

L.32 `push_neg at not_unique,`

L.33 `cases hlm with A hA,` La táctica `cases` nos permite, entre otras cosas, dada una hipótesis de existencia, obtener un término del tipo cuantificado por el existe y la correspondiente hipótesis particularizada para el nuevo término.

En nuestro caso tenemos la hipótesis `hlm: have_common_point Point l m` y la definición `have_common_point Point l m := ∃ A : Point, is_common_point A l m.`

Por tanto al aplicar la táctica, la hipótesis hlm se convierte en dos nuevas hipótesis

```
A : Point
hA: is_common_point A l m
```

L.34 `rcases not_unique A hA with ⟨B, ⟨hB, hAB⟩⟩,` En esta línea están ocurriendo distintas cosas:

- Recordemos que en el estado táctico actual tenemos la hipótesis

```
not_unique: ∀ (x : Point), is_common_point x l m
→ (∃ (y : Point), is_common_point y l m ∧ y ≠ x)
```

  Primero se está construyendo el término `not_unique A hA`, al que posteriormente se le aplicará la táctica rcases.

  En Lean los cuantificadores universales y las implicaciones pueden tratarse como funciones. Al pasar el primer argumento `A` estamos particularizando la cuantificación sobre el punto `x`, proporcionando el término `A : Point` que tenemos entre nuestras hipótesis. Por tanto el término `not_unique A` es igual a

```
is_common_point A l m → (∃ (y : Point), is_common_point y l m ∧ y ≠ A)
```

  Ahora podemos observar que tenemos entre nuestras hipótesis la condición de esta implicación, `hA: is_common_point A l m` Al pasar este término como segundo argumento obtenemos la conclusión de la implicación, y por tanto el término `not_unique A hA` es igual a

```
∃ (y : Point), is_common_point y l m ∧ y ≠ x
```

- La aplicación de la táctica `rcases` nos permite, como anteriormente, obtener un término concreto del cuantificador existencial y además profundizar en la definición recursiva del ∧, generando así dos hipótesis separadas. Obtenemos por tanto las nuevas hipótesis

```
B: Point
hB: is_common_point B l m
hAB: B ≠ A
```

L.35 `rw ne_comm at hAB,` Para tener la hipótesis `hAB: B ≠ A` en el mismo orden que el utilizado en los axiomas y poder utilizarlos correctamente, reescribimos la hipótesis `hAB` utilizando la propiedad conmutativa de la desigualdad, obteniendo así la hipóteis `hAB: A ≠ B`.

L.36 `exact unique_of_exists_unique (ig.I1 hAB) ⟨hA.left,hB.left⟩ ⟨hA.right,hB.right⟩,`

La táctica `exact` se utiliza para concluir la demostración proporcionando un término igual a la meta. Recordemos que la meta actual es `⊢ l = m`.

Analicemos entonces el término que estamos proporcionando a la táctica.

El lema `unique_of_exists_unique`, definido en la librería estándar de Lean, sirve para extrer la parte de unicidad del cuantificador `∃!`. Dadas una fórmula de la forma `∃! x, px` y dos fórmulas `p a` y `p b` , devuelve la fórmula que aserta la igualdad entre los términos que cumplen la propiedad `p: a = b`.

Como primer argumento le estamos pasando el primer axioma de incidencia, particularizado con la hipóteis `hAB : A ≠ B`. Es decir `ig.I1 hAB` es igual a `∃! l : Line, A ~ l ∧ B ~ l`.

Ahora queremos pasar en los otros dos argumentos términos `A ~ l ∧ B ~ l` y `A ~ m ∧ B ~ m`, para obtener la igualdad `l = m`. Para esto tenemos que recombinar las hipótesis `hA` y `hB`.

`hA.left` es igual a `A ~ l` y `hB.left` a `B ~ l`, y mediante los paréntesis ⟨⟩ combinamos estos términos en la conjunción `⟨hA.left,hB.left⟩`, obteniendo `A ~ l ∧ B ~ l`.

El uso de los paréntesis nos permite construir una conjunción sin tener que especificar esplícitamente que queremos construir una conjunción, pero el sistema de tipos de Lean permite inferir que el término esperado es una conjunción.

Análogamente para el segundo argumento.

Pendiente: Mencionar que el código del modo táctico no se parece tanto a la forma usual de escribir matemáticas, pero la experiencia dada por el entorno sí que se acerca más.

De las siguientes proposiciones incluiremos sólo los enunciados. El desarrollo de las demostraciones se encuentra en el repositorio.

Proposición 2. *Tres puntos no colineares determinan tres líneas distintas.*

```
lemma non_collinear_ne_lines
  {Point : Type*} (Line : Type*) [ig: incidence_geometry Point Line]
  (A B C: Point)
  (h_noncollinear : ¬ collinear Line A B C)
  -- Las hipótesis de que los puntos son distintos son innecesarias puesto que
  -- podrían derivarse de `non_collinear_neq`. Están incluidas para darles un
  -- nombre y poder construir las líneas correspondientes en el enunciado.
  (hAB : A ≠ B) (hAC : A ≠ C) (hBC : B ≠ C) :
  (line Line hAB).val ≠ (line Line hAC).val
  ∧ (line Line hAB).val ≠ (line Line hBC).val
  ∧ (line Line hAC).val ≠ (line Line hBC).val :=
 begin
```

src/incidence_geometry/propositions.lean

Proposición 3. *Existen tres líneas distintas que no pasan por un punto común, es decir tales que no existe un punto que está en todas ellas.*

```
lemma exist_neq_lines_not_concurrent
  {Point Line : Type*} [ig : incidence_geometry Point Line] :
  ∃ l m n: Line,
    (l ≠ m ∧ l ≠ n ∧ m ≠ n)
    ∧ ¬ ∃ P : Point, is_common_point P l m
    ∧ is_common_point P l n
    ∧ is_common_point P m n
  :=
begin
```

src/incidence_geometry/propositions.lean

Proposición 4. *Para cada línea existe un punto que no está ella.*

```
lemma line_has_external_point
  {Point Line : Type*} [ig : incidence_geometry Point Line] :
  ∀ l : Line, ∃ A : Point, ¬ A ~ l :=
begin
```

src/incidence_geometry/propositions.lean

Proposición 5. *Para cada punto existe una línea que no pasa por él.*

```
lemma point_has_external_line
  {Point Line : Type*} [ig : incidence_geometry Point Line] :
  ∀ A: Point, ∃ l: Line, ¬ A ~ l :=
begin
```

src/incidence_geometry/propositions.lean

4.2. Geometría del orden

El segundo grupo de axiomas establece propiedades de la relación de *orden*, una relación ternaria entre puntos. Dados tres puntos A, B, C escribiremos $A * B * C$ para indicar que están relacionados mediante la relación de orden. Los cuatro *axiomas de orden* son los siguientes:

Axioma B1. *Si un punto B está entre A y C ($A * B * C$) entonces A, B, C son distintos, están en una misma línea y $C * B * A$.*

Axioma B2. *Para cada dos puntos distintos A, B existe un punto C tal que $A * B * C$.*

Axioma B3. *Dados tres puntos distintos en una línea, uno y sólo uno de ellos está entre los otros dos.*

Axioma B4 (Pasch)**.** *Sean A, B, C tres puntos no colineares y l una línea que no contenga a ninguno de estos puntos. Si l contiene un punto D que está entre A y B ($A * D * B$) entonces también debe contener un punto entre A y C o un punto entre B y C.*

```
class order_geometry (Point Line : Type*)
  extends incidence_geometry Point Line :=
  (between: Point → Point → Point → Prop)
  (notation A `*` B `*` C := between A B C)
  (B11 {A B C: Point} (h : A * B * C) :
    neq3 A B C ∧ collinear Line A B C ∧ C * B * A)
  (B12 {A B C: Point} (h : A * B * C) : C * B * A)
  (B2 {A B : Point} (h : A ≠ B) : ∃ C : Point, A * B * C)
  (B3 {A B C : Point} (h : collinear Line A B C) :
    xor3 (A * B * C) (B * A * C) (A * C * B))
  (B4 {A B C D : Point} {l : Line} (h_non_collinear : ¬ collinear Line A B C)
      (hlABC : ¬ (A ~ l ∨ B ~ l ∨ C ~ l)) (hlD : D ~ l) (hADB : A * D * B)
      : xor (∃ E ,  E ~ l ∧ (A * E * C)) (∃ E, E ~ l ∧ (B * E * C)))
```

src/order_geometry/basic.lean

Pendiente: explicar extension de clases.

Definiciones

Definición (Segmentos). *Dados dos puntos distintos A, B definimos el **segmento** $\overline{AB}$ como el conjunto de puntos que contiene a A, B y a todos los puntos que están entre ellos. Diremos que A y B son los extremos del segmento $\overline{AB}$.*

```
structure Seg (Point : Type*) := {A B : Point} (neq : A ≠ B)
```

src/order_geometry/basic.lean

Definición (Pertenencia de puntos a segmentos). ***Pendiente:*** *redactar*

```
def Seg.in
  {Point : Type*} (seg : Seg Point) (Line : Type*)
  [og : order_geometry Point Line]  (P : Point) : Prop :=
  P = seg.A ∨ P = seg.B ∨ (og.between seg.A P seg.B)
```

src/order_geometry/basic.lean

```
instance [order_geometry Point Line] : has_mem Point (Seg Point) :=
  ⟨λ P seg, seg.in Line P⟩
```

src/order_geometry/basic.lean

Pendiente: Explicar cómo se puede usar la notación

Definición (Intersección entre segmentos y líneas). ***Pendiente:*** *redactar*

```
def seg_intersect_line
  [order_geometry Point Line] (S : Seg Point) (l : Line) :=
  ∃ P : Point, P ∈ S ∧ P ~ l
```

src/order_geometry/basic.lean

Definición (Triángulos). ***Pendiente:*** *redactar*

```
structure Tri (Point Line: Type*) [order_geometry Point Line] :=
  {A B C : Point}
  (non_collinear : ¬ collinear Line A B C)
```

src/order_geometry/basic.lean

Definición (Rayos). *Dados dos puntos distintos A, B, definimos el **rayo** $\overrightarrow{AB}$ como el conjunto que contiene a A y a todos los puntos de la línea AB tales que están del mismo lado de A que B. Dado un rayo $\overrightarrow{AB}$ llamaremos **vértice** del rayo al punto A.*

```
structure Ray (Point : Type*) := {A B: Point} (neq : A ≠ B)
```

src/order_geometry/basic.lean

Definición (Ángulos). *Un* ***ángulo*** *es la unión de dos rayos* $\overrightarrow{AB}$ *y* $\overrightarrow{AC}$ *con el mismo vértice y no contenidos en una misma recta. Denotaremos dicho ángulo por* $\angle ABC$.

```
structure Ang (Point Line: Type*) [order_geometry Point Line] :=
  (r1 r2 : Ray Point)
  (vertex : r1.A = r2.A)
  (non_collinear : ¬ collinear Line r1.A r1.B r2.B)
```

src/order_geometry/basic.lean

```
def Ang.mk_from_points [order_geometry Point Line] (B A C : Point)
  (h : ¬ collinear Line A B C) : Ang Point Line :=
begin
  let neq := non_collinear_neq Line h,
  let r1 := Ray.mk neq.left,
  let r2 := Ray.mk neq.right.left,
  have vertex : r1.A = r2.A, { refl },
  exact ⟨r1, r2, vertex, h⟩
end
```

src/order_geometry/basic.lean

Definición. *Diremos que dos puntos* ***están del mismo lado del plano*** *respecto de una recta si el segmento que los une no contiene ningún punto de la recta.*

Diremos que dos puntos están del mismo ***lado de una recta*** *respecto de un punto si el segmento que los une no contiene a dicho punto.*

```
def same_side_line (l: Line) (A B : Point) :=
  A = B ∨ (∃ h : A ≠ B, ¬ @seg_intersect_line Point Line og (Seg.mk h) l)
```

src/order_geometry/basic.lean

Definición (Lados de una línea). ***Pendiente:*** *redactar*

```
def same_side_point {Point : Type*} (Line : Type*) [order_geometry Point Line]
  (A B C : Point) (hBC : B ≠ C) :=
  collinear Line A B C ∧ A ∉ (Seg.mk hBC)
```

src/order_geometry/basic.lean

Resultados

Pendiente: Escribir sobre el paper de Meikle y los descubrimientos sobre las demostraciones originales de Hilbert

```
lemma point_between_given {A C : Point} (hAC : A ≠ C): ∃ B : Point, A * B * C :=
begin
  sorry
end
```

src/order_geometry/propositions.lean

Proposición 6. *La relación de estar del mismo lado del plano respecto de una recta es una relación de equivalencia.*

Pendiente: ¿Incluir código? Anexo?

4.3. Geometría de congruencia

4.3.1. Para segmentos

Axioma C1. *Dados un segmento $\overline{AB}$ y un rayo r con vértice C, existe un único punto D en el rayo r tal que $\overline{AB} \cong \overline{CD}$.*

Axioma C2. *Si $\overline{AB} \cong \overline{CD}$ y $\overline{AB} \cong \overline{EF}$ entonces $\overline{CD} \cong \overline{EF}$. Además cada segmento es congruente con sí mismo.*

Axioma C3 (Suma)**.** *Dados tres puntos A, B, C en una línea y tales que $A * B * C$ y otros tres puntos D, E, F en una línea tales que $D * E * F$, si $\overline{AB} \cong \overline{DE}$ y $\overline{BC} \cong \overline{EF}$ entonces $\overline{AC} \cong \overline{DF}$.*

4.3.2. Para ángulos

Axioma C4. *Dados un ángulo $\angle ABC$ y un rayo $\overrightarrow{DF}$, fijado un lado del plano de la línea DF, existe un único rayo $\overrightarrow{DE}$ en dicho lado tal que $\angle BAC \cong \angle EDF$.*

Axioma C5. *Para cada tres ángulos α, β, γ, si $\alpha \cong \beta$ y $\alpha \cong \gamma$ entonces $\beta \cong \gamma$. Además cada ángulo es congruente con sí mismo.*

Axioma C6 (SAS)**.** *Sean tres triángulos ABC y DEF tales que $\overline{AB} \cong \overline{DE}$, $\overline{AC} \cong \overline{DF}$ y $\angle BAC \cong \angle EDF$. Entonces los dos triángulos son congruentes, es decir $\overline{BC} \cong \overline{EF}$, $\angle ABC \cong \angle DEF$ y $\angle ACB \cong \angle DFE$.*

```
class congruence_geometry (Point Line : Type*)
  extends order_geometry Point Line :=
  -- Relación de congruencia entre segmentos
  (scong : Seg Point → Seg Point → Prop)
  -- Relación de congruencia entre ángulos
  (acong : Ang Point Line → Ang Point Line → Prop)
  -- Axioma C1: Dados un segmento s y dos puntos distintos A y B, existe un
  -- único punto C del mismo lado de B respecto de A tal que el segmento AC es
  -- congruente a s.
  (C1 (s : Seg Point) (A B : Point) (hAB : A ≠ B) :
    ∃! C : Point, ∃ hBC : B ≠ C, ∃ hAC : A ≠ C,
      (same_side_point Line A B C hBC)
      ∧ (scong s (Seg.mk hAC))
  )
  -- Axioma C2: Dados tres segmentos, si el primero es congruente al segundo y
  -- al tercero, entonces el segundo y tercero son congruentes. Además cada
  -- segmento es congruente con sí mismo.
  (C21 {s1 s2 s3: Seg Point} (h1 : scong s1 s2) (h2 : scong s1 s3) :
    scong s2 s3)
  (C22 (s : Seg Point) : scong s s)
  -- Axioma C3 (Suma de segmentos): Dados tres puntos A, B y C tales que
  -- A * B * C, y otros tres puntos D E F
  (C3 {A B C D E F: Point}
    (hABC : neq3 A B C ∧ between A B C)
    (hDEF : neq3 D E F ∧ between D E F)
    (h1 : scong (Seg.mk hABC.1.1) (Seg.mk hDEF.1.1))
    (h2 : scong (Seg.mk hABC.1.2.2) (Seg.mk hDEF.1.2.2)) :
    scong (Seg.mk hABC.1.2.1) (Seg.mk hDEF.1.2.1)
  )
  (C4 (α : Ang Point Line) (A B Side : Point) (hAB : A ≠ B)
    (hSide :  ¬ Side ~ (line Line hAB).val) :
    ∃! C : Point, ∃ h : same_side_line (line Line hAB).val Side C,
      (acong α (Ang.mk_from_points B A C
        (same_side_line_non_collinear hAB hSide h))))
  (C51 {α β γ : Ang Point Line} (hαβ : acong α β) (hαγ : acong α γ) :
    acong β γ)
  (C52 (α : Ang Point Line) : acong α α)
  (C6 (T1 T2 : Tri Point Line)
    (h1: scong (Seg.mk T1.neq.1) (Seg.mk T2.neq.1))
    (h2: scong (Seg.mk T1.neq.2.1) (Seg.mk T2.neq.2.1))
    (h3: acong (Ang.mk_from_points T1.B T1.A T1.C T1.non_collinear)
               (Ang.mk_from_points T2.B T2.A T2.C T2.non_collinear)) :
      scong (Seg.mk T1.neq.2.2) (Seg.mk T2.neq.2.2)
      ∧ acong
        (Ang.mk_from_points T1.A T1.B T1.C T1.non_collinear_symm)
        (Ang.mk_from_points T2.A T2.B T2.C T2.non_collinear_symm)
      ∧ acong
        (Ang.mk_from_points T1.A T1.C T1.B T1.non_collinear_symm')
        (Ang.mk_from_points T2.A T2.C T2.B T2.non_collinear_symm')
  )
```

src/congruence_geometry/basic.lean

Referencias

[AvD] Jeremy Avigad y Floris van Doorn. «The Lean Theorem Prover and Homotopy Type Theory». En: ().

[Gre93] Marvin J. Greenberg. *Euclidean and Non-Euclidean Geometries: Development and History*. 3rd ed. New York: W.H. Freeman, 1993. 483 págs. ISBN: 978-0-7167-2446-9.

[Har00] Robin Hartshorne. *Geometry: Euclid and Beyond*. Red. de S. Axler, F. W. Gehring y K. A. Ribet. Undergraduate Texts in Mathematics. New York, NY: Springer New York, 2000. ISBN: 978-1-4419-3145-0 978-0-387-22676-7. DOI: 10.1007/978-0-387-22676-7. URL: http://link.springer.com/10.1007/978-0-387-22676-7 (visitado 27-03-2023).

referencias