

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 que implementa una teoría de tipos, el *Cálculo de construcciones inductivas*, utilizando la axiomática de Hilbert de la geometría euclídea como guía motivadora para el aprendizaje.

Comentamos aspectos generales de la formalización de matemáticas asistida por computadoras, sus aplicaciones y ventajas; se introducen conceptos básicos del funcionamiento de Lean; se analiza el trabajo de formalización realizado; y se concluye proponiendo una línea de continuación del trabajo.

Abstract

This paper explores the use of the Lean proof assistant, a language that implements a type theory, the *Calculus of Inductive Constructions*, using Hilbert's axiomatics of Euclidean geometry as a motivating guide for learning.

We discuss the general aspects of the computer-assisted formalization of mathematics, its applications and advantages; we also introduce basic concepts of Lean operation; we analyze the formalization work carried out; and we conclude by proposing a line of continuation of the work.

Índice

1. Objetivos y metodología	3
2. Matemáticas y geometría formales	4
3. Formalización asistida por computadores	5
4. Introducción a Lean	7
4.1. Teoría de tipos informal	7
4.2. Funciones en el λ -cálculo	8
4.3. Introducción de términos	10
4.4. Proposiciones	11
4.5. Demostraciones	11
4.6. <i>mathlib</i>	13
5. Formalizando la geometría de Hilbert en Lean	13
5.1. Geometría de incidencia	14
5.2. Geometría del orden	22
5.3. Geometría de congruencia	29
5.4. El axioma de las paralelas	31
6. Conclusiones	33
A. Repositorio y organización del código	36
B. Entorno de desarrollo	36
Referencias	38
Agradecimientos	40

1. Objetivos y metodología

El objetivo principal de este trabajo es aprender los fundamentos del lenguaje de programación y asistente de demostración Lean. Sin entrar en detalles sobre cómo funciona y la implementación del lenguaje, nos hemos propuesto aprender a utilizarlo como podría usarlo un estudiante o investigador de matemáticas. Es decir, formalizando resultados matemáticos.

Para esto hemos elegido la geometría euclídea plana, en particular la axiomatización de David Hilbert realizada a inicios del siglo pasado. La idea es que esta teoría sirva de guión para formalizar resultados y durante el proceso de formalización de axiomas y proposiciones aprender las características y nociones necesarias de Lean.

Otro objetivo más ambicioso, que trasciende las posibilidades de este trabajo, es el de formalizar la independencia del *axioma de las paralelas* del resto de axiomas de la geometría. Aún no teniendo el tiempo suficiente para abordar esta tarea, hemos propuesto una formalización del enunciado de la independencia y propuesto un plan de trabajo para formalizar su demostración.

Antes de poder emprender el trabajo de formalización de resultados geométricos ha sido necesario obtener unos conocimientos de base de Lean. Al abordar esta tarea, en la que aprendimos a expresar y demostrar resultados elementales de lógica proposicional, a trabajar con estructuras algebraicas y exploramos ciertas características del lenguaje, las principales referencias han sido los materiales del curso de Kevin Buzzard, *Formalising mathematics* [1], y el manual oficial, *Theorem proving in Lean* [2].

Los materiales del curso, que incluyen ejercicios prácticos, explicaciones teóricas y soluciones a los ejercicios en formato vídeo han sido especialmente útiles en este proceso de aprendizaje.

Durante el mes de febrero de 2023 tuvimos la oportunidad de participar en el curso de doctorado sobre *Formalización de matemáticas en Lean*, impartido por María Inés de Frutos [3]. Asistir a este curso nos permitió afianzar los conocimientos que estábamos adquiriendo y contrastar dudas o distintas formas de resolver los problemas con la profesora y los demás participantes del curso.

Para tener un entendimiento a nivel intuitivo de los conceptos necesarios de teoría de tipos se ha estudiado el primer capítulo del libro *Homotopy Type Theory* [4] en el que se introducen conceptos de teoría de tipos y se mencionan sus relaciones con la lógica.

En paralelo a este aprendizaje del lenguaje y herramientas de Lean iniciamos el estudio de la geometría de Hilbert, apoyándonos principalmente en los libros de texto de Hartshorne [5] y Greenberg [6], y realizando alguna consulta al texto original de Hilbert [7], para conocer cómo fueron formulados originalmente los axiomas, definiciones y teoremas de la teoría.

Una vez abordados los conceptos básicos necesarios de Lean y de la geometría euclídea, emprendimos la tarea de la formalización. Para ello, el proceso ha consistido fundamentalmente en seguir linealmente la teoría como está presentada en los libros de referencia e intentar formalizar axiomas, definiciones, proposiciones y demostraciones. A medida que hemos ido avanzando y la dificultad de los resultados por formalizar ha ido creciendo, hemos tenido que volver a las referencias de Lean, consultando documentación y aprendiendo nuevas características del lenguaje.

El código completo que hemos desarrollado puede encontrarse en un repositorio github. En el apéndice [A](#) se encuentra el enlace y un esquema de la estructura de los ficheros. En este documento, cada vez que incluimos extractos de estos ficheros hemos referenciado las líneas incluidas y nombre del fichero.

Durante este proceso de digitalización hemos consultado trabajos similares que analizan la geometría de Hilbert, uno realizado en el lenguaje Isabelle/Isar [8], en el que se analizan en detalle las demostraciones originales de Hilbert, y otro que utiliza el lenguaje Coq/Gallina [9], en el que se exploran formalizaciones mediante una lógica intuicionista.

2. 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. De los axiomas se pueden deducir, mediante la aplicación sistemática de las reglas de la lógica, nuevas proposiciones matemáticas. El proceso de aplicar secuencialmente estas reglas 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* 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*) [7], 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.

3. Formalización asistida por computadores

En la docencia e investigación en matemáticas es usual escribir las demostraciones utilizando el lenguaje natural. Se asume que cada paso podría escribirse en un lenguaje formal, en cuyo caso podría comprobarse que cada paso deriva de los anteriores y la aplicación de las reglas lógicas. Este trabajo es demasiado tedioso para los matemáticos, y el exceso de detalles puede restar protagonismo a las ideas que se quieren exponer.

Sin embargo es una tarea perfecta para ser realizada de forma automática por máquinas. Los asistentes de demostración nos permiten formalizar definiciones, enunciados de teoremas, sus demostraciones, y verificar la corrección de estas pruebas. Algunos de estos sistemas son Coq, Isabelle/HOL, Agda, Metamath, Mizar y Lean.

Además de un lenguaje, estos sistemas suelen constar de un entorno de desarrollo que aporta información contextual útil en el proceso de demostración. Lean es especialmente popular debido a la calidad y facilidad de uso de este entorno (ver apéndice B) y a la existencia de *mathlib*, una amplia librería de matemáticas formalizadas.

Formalizar matemáticas con la ayuda de un asistente de demostraciones consiste en digitalizar enunciados y resultados escribiéndolos en un lenguaje de programación que garantiza, mediante una correspondencia entre una teoría de tipos y la lógica con el lenguaje de la teoría de conjuntos, la validez de cada paso.

En lo que sigue de esta sección expondremos algunos beneficios y aplicaciones de los asistentes de demostración. La conferencia *¿Por qué formalizar matemáticas?* impartida por María Inés de Frutos en la facultad de Ciencias Matemáticas de la UCM ha sido la principal referencia a la hora de elaborar esta lista [10].

Comprobación mecanizada de demostraciones. Estos sistemas pueden verificar la corrección de demostraciones largas y técnicas, reduciéndolas a las aplicaciones fundamentales de las reglas del sistema formal utilizado, como una teoría de tipos.

Un ejemplo en el que se ha conseguido formalizar un resultado muy avanzado es el del *Liquid Tensor Experiment*. En el año 2020 el medallista Fields Peter Scholze lanzó a la comunidad de formalización matemática el reto de verificar un importante teorema publicado por él y Dustin Clausen [11]. Scholze explica que, antes de que se formalizara el resultado, no estaba totalmente seguro de la corrección de una parte técnica de la demostración, que contenía muchos detalles técnicos que la comunidad no estaba estudiando.

La guía de esta demostración, que se terminó de formalizar en el año 2022, se encuentra en formato web [12] y es un buen ejemplo de un documento que presenta resultados paralelamente en un formato textual, legible por matemáticos, y su formalización en el lenguaje Lean.

Una de las visiones a futuro de los impulsores de Lean, como Kevin Buzzard, es la idea de a cada artículo matemático se le asocie su formalización y por tanto el papel de los revisores pase a enfocarse en el interés de los resultados, puesto que su corrección estaría garantizada por el asistente de demostración. Actualmente este objetivo está aún lejos de ser alcanzado.

Digitalización de definiciones y enunciados. La creación de una base de datos digital puede mejorar la capacidad de búsqueda de resultados matemáticos previos, convirtiéndose en un gran apoyo en el proceso de investigación. El proyecto *Formal Abstracts* [13] se propone realizar una enciclopedia matemática digital utilizando el lenguaje Lean, con el objetivo de formalizar la mayor cantidad posible de teoremas publicados en un formato entendible a la vez por humanos y máquinas. Además se quieren enlazar los términos utilizados en dichos enunciados con sus definiciones precisas.

El proyecto se encuentra en fase de diseño y queda bastante trabajo para que alcance sus objetivos, pero muestra una dirección de trabajo muy interesante.

Uso en docencia. En ciertos centros universitarios ya se imparten cursos apoyándose en el uso del asistente Lean. El principal impulsor del uso de estas herramientas en la docencia es Kevin Buzzard, que desde el año 2021 imparte el curso *Formalising mathematics* [1] en el Imperial College.

El uso de asistentes de demostración en la docencia fija una referencia objetiva sobre qué es una demostración correcta de un resultado, obligando a los estudiantes a ser precisos en sus razonamientos. Además los estudiantes pueden aprovechar el entorno interactivo para comprobar autónomamente si sus razonamientos son correctos, sin necesidad de esperar la corrección del profesor.

El principal problema actual en este ámbito es que la barrera de entrada al uso de estos asistentes es alta, puesto que la instalación de las herramientas y librerías no es simple y es necesaria cierta familiaridad con el lenguaje para escribir código.

Demostración automatizada. Se espera que en un futuro se puedan incorporar a estos sistemas técnicas de inteligencia artificial que proporcionen indicaciones y ayuda a los matemáticos en el proceso de demostración. Un ejemplo de avance en este campo es el artículo [14] publicado recientemente por la empresa OpenAI, en el que se han conseguido automatizar algunas demostraciones de olimpiadas de matemáticas.

4. Introducción a Lean

4.1. Teoría de tipos informal

Lo primero que se necesita en matemáticas para formalizar afirmaciones es un lenguaje formal con el cual expresarlas. Normalmente se utiliza la lógica de primer orden con los axiomas de la teoría de conjuntos. Lean¹, sin embargo, utiliza un sistema deductivo diferente, el de la teoría de tipos *Cálculo de construcciones inductivas* (*Calculus of Inductive Constructions* en inglés).

En lógica se tiene la aserción fundamental «que de una serie de fórmulas se pueda derivar una demostración de una proposición». Es decir, cada proposición P da lugar a la aserción correspondiente « P tiene una prueba» (a veces dejamos implícito el contexto de fórmulas desde las que se deriva la demostración). 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».

¹En este trabajo hemos utilizado a la versión 3 del lenguaje Lean. La versión más actual del lenguaje es la 4, que incorpora muchos cambios. Gran parte de la comunidad sin embargo continúa utilizando la versión 3, mientras que se terminan de migrar los resultados ya formalizados a la nueva versión.

En una teoría de tipos la aserción fundamental es «que de una serie de términos con sus tipos se pueda derivar que un término tiene un tipo»². Dados un término a y un tipo A , si a tiene tipo A escribimos $a : A$. Esta es misma notación es la utilizada en Lean, en el que por ejemplo podemos expresar la aserción «3 es un número natural» con el código $3 : \mathbb{N}$. 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 las teorías de tipos no podemos considerar los términos o los tipos por separado, cada término tiene que estar siempre acompañado por su tipo. En las teorías 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 $a \equiv b : A$ ³.

Las teorías de tipos también pueden utilizarse para expresar afirmaciones y pruebas 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 una 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.

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 *Cálculo de las construcciones inductivas*.

En las explicaciones que siguen la principal referencia a la que hemos recurrido es el manual oficial *Theorem proving in Lean* [2].

4.2. Funciones en el λ -cálculo

La base de muchas teorías de tipos es el lambda cálculo tipado. El lambda cálculo es 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 $x + 1 : \mathbb{N}$, se puede convertir la variable libre x en en una variable ligada por la abstracción, a la que llamamos parámetro de la función, mediante la sintaxis $\lambda x : \mathbb{N}, (x + 1 : \mathbb{N})$.

²La referencia que hemos seguido en esta introducción a la teoría de tipos es el primer capítulo del libro *Homotopy Type Theory* [4].

³En Lean existe más de un tipo de igualdad entre términos. La mencionada aquí es la llamada igualdad *sintáctica*, que se representa con el símbolo $=$. También existen las igualdades *definicionales* y *proposicionales*.

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 \mathbb{N} . Es decir, tenemos una función que dado un número natural devuelve otro número natural. Esto también puede escribirse como $(\lambda x, x + 1) : \mathbb{N} \rightarrow \mathbb{N}$.

Lean además incluye notación para definir funciones que devuelven otras funciones, lo cual es muy útil para tratar funciones que reciben más de un parámetro⁴. Las siguientes líneas de código definen expresiones equivalentes que representan el mismo término, de tipo $\alpha \rightarrow \beta \rightarrow \alpha$.

```
 $\lambda a : \alpha, \lambda b : \beta, a$   
 $\lambda (a : \alpha) (b : \beta), a$ 
```

- **Evaluación.** Consiste en aplicar funciones, pasándoles los valores de los argumentos que evaluar. Por ejemplo la expresión

```
 $(\lambda x : \mathbb{N}, (x + 1) : \mathbb{N}) (1 : \mathbb{N})$ 
```

indica que estamos evaluando la función $(\lambda x, x + 1) : \mathbb{N} \rightarrow \mathbb{N}$ con el parámetro x sustituido por el término $1 : \mathbb{N}$ (para que la sustitución pueda realizarse los tipos deben coincidir). Mediante un proceso de reducción se obtiene el término $2 : \mathbb{N}$.

La teoría de tipos que implementa Lean, el *Cálculo de las construcciones inductivas*, introduce una serie de reglas y formas de construir tipos adicionales, como los *tipos inductivos* o los *tipos dependientes*, útiles para formalizar definiciones inductivas y cuantificadores lógicos.

Como se ve en los ejemplos, en el código fuente de Lean se pueden incluir caracteres unicode, como λ , \rightarrow o \mathbb{N} . En el apéndice B explicamos cómo introducir estos caracteres en el entorno de desarrollo.

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 $\lambda x : \mathbb{N}, (x + 1) : \mathbb{N}$ no es necesario incluir la segunda anotación de tipo. Dada la expresión $x + 1$ y sabiendo por contexto que $x : \mathbb{N}$ 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 \mathbb{N} .

⁴La transformación de funciones de varios parámetros en funciones de orden superior se denomina *curriificación*.

4.3. 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

También se pueden definir nuevos términos introduciendo nombres, en este caso sin introducir nuevos axiomas, 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 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 automáticamente.

Variables

Otra forma de indicar parámetros en una definición es mediante el uso del comando `variable` (o su variante `variables`) que introduce en el entorno el símbolo indicado, pero en lugar de fijar su existencia como `constant` el símbolo introducido está «cuantificado universalmente». Las siguientes líneas son equivalentes a las del ejemplo anterior:

```
variable {α : Type*}  
def id3 (e : α) := e
```

4.4. 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, que relaciona *tipos* con *proposiciones*. Los términos de tipo `P : Prop`, que representan las proposiciones, son a su vez tipos y por tanto pueden tener términos.

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

También tenemos a nuestra disposición los demás conectores lógicos usuales, definidos en la librería estándar de Lean. Los símbolos que utilizaremos para estos conectores son \wedge , \vee , \neg , \neq y \leftrightarrow . Además, tipos del *Cálculo de construcciones inductivas* como el *tipo suma* y *tipo producto* permiten definir los cuantificadores existencial y universal, expresados en Lean mediante los símbolos \exists y \forall .

4.5. Demostraciones

Mientras que en la correspondencia de Curry-Howard las proposiciones se corresponden con los tipos, la demostración de una proposición se corresponde con el hecho de que el tipo de la proposición tenga un término. Es decir, dada una proposición `P : Prop`, un término `p : P` es una demostración de dicha proposición.

Si un tipo tiene un término se dice que está *habitado*. Por tanto decir que un término de tipo `Prop` está habitado equivale a decir que la proposición correspondiente tiene una demostración. Esta equivalencia entre términos y demostraciones es llamada *demostraciones como términos*.

Sin embargo desarrollar demostraciones construyendo directamente estos términos suele resultar muy complicado y alejado del trabajo deductivo matemático usual. Como muchos otros asistentes de demostración Lean tiene definidos una serie de comandos, llamados *tácticas*, que permiten describir en un nivel de abstracción más alto cómo se construyen estas demostraciones.

En matemáticas solemos recurrir a frases informales como «podemos demostrar la siguiente afirmación aplicando el lema anterior, deshaciendo las definiciones y simplificando la expresión resultante», que proporcionan una «receta» de construcción de demostración. De forma análoga las tácticas son instrucciones que indican al sistema de tipos de Lean cómo construir el término de una demostración.

En Lean se pueden desarrollar demostraciones utilizando los denominados *modo de términos* y *modo táctico*, que además pueden ser entremezclados. En este trabajo nos hemos limitado a utilizar el *modo táctico*, ya que resulta más intuitivo, fácil de usar y cercano a la forma de demostrar usual en matemáticas.

Para enunciar un resultado y demostrarlo se utiliza el comando `lemma` o `theorem`, con la misma sintaxis de `def`:

```
lemma ej (P Q : Prop) (h1 : P ∨ Q) (h2 : P → Q) : Q :=
  or.elim h1 h2 (λ hQ : Q, hQ)
```

En este ejemplo se ha utilizado el modo de términos para desarrollar la demostración. La misma demostración se puede realizar en el modo táctico, al que podemos entrar usando las instrucciones `begin` y `end`:

```
lemma ej (P Q : Prop) (h1 : P ∨ Q) (h2 : P → Q) : Q :=
begin
  cases h1,
  sorry
end
```

Una demostración en modo táctico consiste en la aplicación sucesiva de tácticas, que van modificando el *estado táctico*, que está formado por una serie de *hipótesis* y una *meta*. Antes de la aplicación de la primera táctica el estado táctico está formado por las hipótesis del lema o teorema y la meta se corresponde con el resultado a demostrar. Modificando las hipótesis y la meta sucesivamente podremos *cerrar la meta*, es decir completar la demostración.

En el ejemplo anterior se ha usado la táctica `sorry`, que cierra cualquier meta y se usa para indicar que todavía no hemos terminado la demostración. Un ejemplo de demostración completa de este ejemplo en modo táctico es:

```
lemma ej (P Q : Prop) (h1 : P ∨ Q) (h2 : P → Q) : Q :=
begin
  cases h1,
  { apply h2, exact h1 },
  { exact h1 },
end
```

4.6. *mathlib*

Uno de los mayores puntos fuertes del ecosistema de Lean es la librería *mathlib*, donde se encuentra una amplia colección de resultados matemáticos formalizados. Además la librería incluye definiciones de tácticas que implementan esquemas de demostración particulares de distintas ramas de las matemáticas.

Debido a la naturaleza del lenguaje y a las particularidades de la teoría de tipos que se utiliza, diversas teorías matemáticas están implementadas de forma distinta a la de muchos libros de texto o a la que se enseña en cursos introductorios del grado de matemáticas. Por ejemplo las construcciones en topología se basan en los *filtros*, en lugar de en los *entornos* y *bases de entornos*. En cualquier caso en la librería es común que estén demostradas las equivalencias con axiomatizaciones y definiciones equivalentes.

5. Formalizando la geometría de Hilbert en Lean

En esta sección presentamos 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). Dicho 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 hemos seguido una versión modernizada de los axiomas y los resultados, basada en las presentaciones de los libros de Hartshorne [5] y Greenberg [6], en las que se considera el caso restringido de la geometría plana.

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

Trataremos 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.

5.1. 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 colineales. 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. Las *estructuras*, implementadas mediante *tipos inductivos*, permiten agrupar varios términos con sus tipos en una misma definición. Como las definiciones, las estructuras pueden estar parametrizadas⁵.

Las clases son estructuras con una funcionalidad añadida, el soporte de inferencia de clases. Mediante este mecanismo podemos declarar *instancias* de clases, que son formas de asignar estructura a objetos.

Veamos un ejemplo. Esta es la definición de clase mediante la que hemos digitalizado las nociones y conceptos de la *geometría de incidencia*:

⁵En la sección 5.2.1 se encuentran varias definiciones que hacen uso de estructuras

```

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 librería *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 lo mismo en el caso de las demostraciones).

Declarar una *instancia* de esta clase consiste en dotar de la estructura que proporciona la clase a algún tipo concreto. De esta forma cuando utilicemos el tipo sobre el que hemos definido esta estructura el sistema de inferencia de tipos de Lean podrá inferir que nuestro tipo tiene la estructura definida en la clase. Por ejemplo si tenemos dos tipos que representan los puntos y líneas de \mathbb{R}^2 podemos dotarles de la estructura definida en la clase `incidence_geometry` mediante el comando `instance`⁶. Para ello tendremos que definir la relación de incidencia `lies_on` entre estos dos tipos y demostrar que se cumplen los tres axiomas.

Podemos pensar en las clases como en descripciones de familias de tipos (para los que existe una instancia de la clase). Para especificar en los parámetros de una definición que uno o más tipos pertenecen a una clase dada se utilizan los corchetes: `[incidence_geometry Point Line]`.

5.1.1. Definiciones

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

⁶No hemos entrado en detalles sobre el uso de este comando debido a la extensión necesaria para definir los tipos mencionados para los puntos y líneas. En el fichero `src/examples.lean` del repositorio se puede consultar el código de este ejemplo.

En las demostraciones es útil tener una forma de, dados dos puntos distintos, construir el término de la línea que pasa por ellos, aprovechando el axioma [I1](#).

```

30 noncomputable def line
31   {Point : Type*} (Line : Type*) [incidence_geometry Point Line]
32   {A B : Point} (h : A ≠ B) :
33   { l : Line // A ~ l ∧ B ~ l } :=
34 begin
35   let hAB := I1 h,
36   rw exists_unique at hAB,
37   let P := λ l : Line, A ~ l ∧ B ~ l,
38   have h1P : ∃ l : Line, P l, { tauto },
39   exact classical.indefinite_description P h1P,
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 (Colinealidad). Decimos que tres puntos distintos son **colineales** si existe una línea que los contiene (todos los puntos inciden en la línea).

```

60 def collinear {Point : Type*} (Line: Type*) [incidence_geometry Point Line]
61   (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.

```

173 def is_common_point
174   {Point Line : Type*} [incidence_geometry Point Line]
175   (A : Point) (l m : Line) :=
176   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.

```

179 def have_common_point
180   (Point : Type*) {Line : Type*} [incidence_geometry Point Line]
181   (l m : Line) :=
182    $\exists$  A : Point, is_common_point A l m

```

src/incidence_geometry/basic.lean

5.1.2. Resultados

Uno de los primeros resultados que se pueden demostrar, utilizando sólo 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. \square

Esta demostración, que sigue la presentación del libro de Hartshorne [5], 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 líneas 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    $\forall$  l m : Line, l  $\neq$  m  $\rightarrow$ 
24     ( $\exists$ ! A : Point, is_common_point A l m)
25      $\vee$  ( $\neg$  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`. Esto equivale a decir en lenguaje natural "sean `l` y `m` dos líneas". La nueva meta es

$$\vdash l \neq m \rightarrow (\exists! (A : \text{Point}), \text{is_common_point } A \ l \ m) \vee \neg \text{have_common_point } \text{Point } l \ m$$

L.28 `contrapose`, La táctica `contrapose` permite recurrir al esquema de la contraposición. Es decir, si nuestra meta es de la forma $A \rightarrow B$, la reemplaza por $\neg B \rightarrow \neg A$. En este caso la meta resultante es

$$\vdash \neg((\exists! (A : \text{Point}), \text{is_common_point } A \ l \ m) \vee \neg \text{have_common_point } \text{Point } l \ m) \rightarrow \neg l \neq m$$

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

$$\vdash (\neg \exists! (A : \text{Point}), \text{is_common_point } A \ l \ m) \wedge \text{have_common_point } \text{Point } l \ m \rightarrow l = m$$

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 $\exists!$. 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 profundizar en definiciones recursivas, en este caso la del operador \wedge , e introducir entre las hipótesis los dos lados de la conjunción, mediante el uso

de los paréntesis $\langle \rangle$. 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 $\vdash l = m$.

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 $A \leftrightarrow B$. Al escribir `at` indicamos dónde queremos realizar dicha reescritura, en este caso en la hipótesis `not_unique`. Como estamos utilizando la definición de $\exists!$, la hipótesis resulta

```
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,`

Una vez reescrita la definición de $\exists!$ podemos continuar empujando la negación hacia dentro de la fórmula. La hipótesis `not_unique` cambia a

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

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 construimos el término `not_unique A hA`, al que luego aplicaremos 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 las definiciones de `∃` y `∧`, generando así tres 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 hipótesis `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 extraer 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 hipótesis `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 $A \sim 1$ y `hB.left` a $B \sim 1$, y mediante los paréntesis $\langle \rangle$ combinamos estos términos en la conjunción $\langle hA.left, hB.left \rangle$, obteniendo $A \sim 1 \wedge B \sim 1$.

El uso de los paréntesis nos permite construir una conjunción sin tener que especificar explícitamente que queremos construir una conjunción. El sistema de inferencia de tipos de Lean deduce que el término esperado es una conjunción.

El segundo argumento que se le pasa al término `ig.I1 hAB` se construye de forma análoga.

Como se puede ver el código que escribimos para realizar una demostración dentro del modo táctico no siempre es fácil de seguir y para entenderlo hay que tener en cuenta cómo va cambiando el estado táctico.

En el entorno de desarrollo (ver apéndice B) se puede consultar de forma interactiva el estado táctico de cada punto de una demostración, lo que facilita enormemente el entendimiento de los pasos de la demostración.

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

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

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

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.*

```
76 lemma exist_neq_lines_not_concurrent
```

src/incidence_geometry/propositions.lean

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

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

src/incidence_geometry/propositions.lean

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

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

src/incidence_geometry/propositions.lean

5.2. 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 esta relación.

Así empieza por tanto la definición de la clase que engloba los axiomas de orden:

```
21 class order_geometry (Point Line : Type*)
22   extends incidence_geometry Point Line :=
23   (between : Point → Point → Point → Prop)
24   (notation A `` B `` C := between A B C)
```

src/order_geometry/basic.lean

Se puede ver en la línea 22 que esta definición de clase *extiende* la de la clase `incidence_geometry` definida anteriormente. Es decir, seguimos considerando la relación de incidencia y los axiomas relativos a ella.

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 $C * B * A$.*

```
25 (B11 {A B C : Point} (h : A * B * C) :
26   neq3 A B C ∧ collinear Line A B C ∧ C * B * A)
27 (B12 {A B C : Point} (h : A * B * C) : C * B * A)
```

src/order_geometry/basic.lean

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

```
28 (B2 {A B : Point} (h : A ≠ B) : ∃ C : Point, A * B * C)
```

src/order_geometry/basic.lean

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

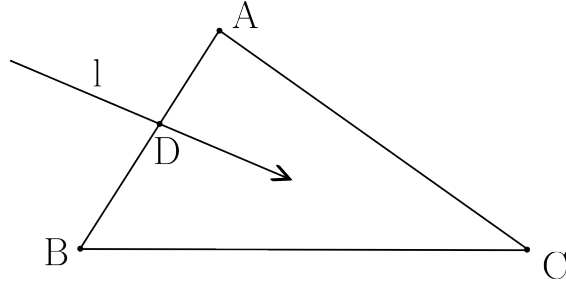
```

29 (B3 {A B C : Point} (h : collinear Line A B C) :
30   xor3 (A * B * C) (B * A * C) (A * C * B))

```

src/order_geometry/basic.lean

Axioma B4 (Pasch). Sean A, B, C tres puntos no colineales 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 .



```

31 (B4 {A B C D : Point} {l : Line} (h_non_collinear : ¬ collinear Line A B C)
32   (hlABC : ¬ (A ~ l ∨ B ~ l ∨ C ~ l)) (hID : D ~ l) (hADB : A * D * B)
33   : xor (∃ E, E ~ l ∧ (A * E * C)) (∃ E, E ~ l ∧ (B * E * C)))

```

src/order_geometry/basic.lean

5.2.1. Definiciones

La definición usual de segmentos es la siguiente:

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.

Pero en nuestro caso no queremos utilizar teoría de conjuntos y por tanto queremos evitar nociones como «conjunto de puntos». Por tanto decimos simplemente que

Definición (Segmentos). Dos puntos distintos A, B determinan el **segmento** \overline{AB} . Diremos que A y B son los **extremos** del segmento \overline{AB} .

Para formalizar esta noción utilizamos una estructura:

```

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

```

src/order_geometry/basic.lean

Cuando se define una estructura en Lean se genera de forma automática un constructor, una función útil para crear términos con tipo la estructura definida. Dicha función se llamará como la estructura más la notación `.mk` adjuntada: en este caso se tiene la función `Seg.mk : (neq : A ≠ B) → Seg Point` (observemos cómo los términos `A B : Point` son implícitos en la definición de la estructura).

Esta definición es suficiente para determinar un segmento, pero para establecer la relación de pertenencia de puntos a segmentos tendremos que complementarla con otra definición.

Definición (Pertenencia de puntos a segmentos). Decimos que un punto A pertenece a un segmento \overline{BC} si A es igual a uno de los extremos del segmento (B o C) o está entre ellos ($B * A * C$).

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

src/order_geometry/basic.lean

Esta función `Seg.in` está dentro del espacio de nombres definido en la estructura `Seg`. Gracias a esto, si tenemos un término de la estructura `seg : Seg Point`, podremos llamar a la función utilizando la notación de punto `seg.in Line P`, en la que el segmento `seg` es pasado como primer argumento a la función `seg`. Por tanto con esta definición y la notación del punto estamos escribiendo la pertenencia de puntos a segmentos en el orden contrario al usual: `seg.in Line P` significa que el punto P pertenece al segmento `seg`. Esta notación del punto también permite acceder a los elementos de una estructura⁷: En la definición de `Seg.in` se puede ver como accedemos a los extremos mediante la notación `seg.A`, `seg.B`.

Además es importante observar que A pertenece a \overline{BC} si y sólo si pertenece a \overline{CB} , lo que si consideráramos la definición basada en conjuntos de puntos equivaldría a decir que los segmentos \overline{BC} y \overline{CB} son iguales. Dicho de otra forma, el orden de los extremos de un segmento no determina la relación de pertenencia de puntos a dichos segmentos.

```
73 lemma seg_mem_symm
74   {Point : Type*} (Line : Type*) [og : order_geometry Point Line]
75   {A B : Point} (P : Point) (hAB : A ≠ B):
76   P ∈ Seg.mk hAB ↔ P ∈ Seg.mk hAB.symm:=
```

src/order_geometry/basic.lean

Como en el caso de los segmentos, las siguientes definiciones hacen referencia a nociones propias de la teoría de conjuntos. En nuestra formalización hemos evitado

⁷Si consideramos la estructura como un producto cartesiano de los tipos que la forman, este acceso se puede ver como una proyección

dichas nociones, por lo que daremos las definiciones adaptadas, utilizando sólo los tipos definidos anteriormente.

Definición (Triángulos). Tres puntos no colineales (por tanto distintos) A , B y C , determinan el **triángulo** $\triangle ABC$. Los puntos A , B y C se llaman **vértices** del triángulo.

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

src/order_geometry/basic.lean

En este caso también podemos definir funciones mediante la notación de punto que nos permitan recuperar propiedades sobre una estructura, como por ejemplo la propiedad de que los vértices del triángulo son distintos:

```
135 lemma Tri.neq
136   {Point Line: Type*} [order_geometry Point Line] (T : Tri Point Line) :
137   neq3 T.A T.B T.C :=
138   by exact non_collinear_neq Line T.non_collinear
```

src/order_geometry/basic.lean

Definición (Pertenencia de puntos a triángulos). Decimos que un punto P pertenece al triángulo $\triangle ABC$ si pertenece a alguno de los segmentos determinados por sus vértices.

```
142 def Tri.in {Point Line: Type*} [order_geometry Point Line]
143   (T: Tri Point Line) (P : Point) : Prop :=
144   (Seg.mk T.neq.1).in Line P
145   ∨ (Seg.mk T.neq.2.1).in Line P
146   ∨ (Seg.mk T.neq.2.2).in Line P
```

src/order_geometry/basic.lean

Como esta definición se basa en la pertenencia de puntos a segmentos, el orden en el que consideremos los puntos del triángulo tampoco será determinante para establecer esta relación de pertenencia.

Definición (Rayos). Decimos que dos puntos distintos A, B definen el **rayo** \overrightarrow{AB} . Dado un rayo \overrightarrow{AB} llamaremos **vértice** del rayo al punto A .

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

src/order_geometry/basic.lean

Definición. Decimos que un punto P pertenece a un rayo \overrightarrow{AB} si coincide con su vértice A o si $A * B * P$.

```

154 def Ray.in (ray : Ray Point) (Line : Type*)
155   [og : order_geometry Point Line] (P : Point) : Prop :=
156   P = ray.A ∨ (ray.A * ray.B * P)

```

src/order_geometry/basic.lean

En este caso el orden de los puntos que determinan un rayo sí que es importante. No se tiene como antes que P pertenece a \overrightarrow{AB} si y sólo si pertenece a \overrightarrow{BA} . En términos conjuntistas tendríamos que los rayos \overrightarrow{AB} y \overrightarrow{BA} son distintos.

Definición (Ángulos). Dos rayos \overrightarrow{AB} y \overrightarrow{AC} con el mismo vértice y tales que A , B y C no están alineados determinan un ángulo. El vértice de los rayos que determinan el ángulo se llama **vértice** del ángulo. Denotaremos dicho ángulo por $\angle ABC$.

```

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

```

src/order_geometry/basic.lean

Como en la definición de ángulo estamos considerando dos rayos con un punto en común, se puede observar que tres puntos no alineados determinan un ángulo. Podemos definir de esta manera un otro constructor para ángulos:

```

184 def Ang.mk_from_points [order_geometry Point Line] (B A C : Point)
185   (h : ¬ collinear Line A B C) : Ang Point Line :=
186   begin
187     let neq := non_collinear_neq Line h,
188     let r1 := Ray.mk neq.left,
189     let r2 := Ray.mk neq.right.left,
190     have vertex : r1.A = r2.A, { refl },
191     exact ⟨r1, r2, vertex, h⟩
192   end

```

src/order_geometry/basic.lean

Observemos que en esta definición hemos utilizado el modo táctico en una definición, lo cual es un recurso útil para construir términos complejos.

Definición. Decimos 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.

```

198 def same_side_line (l: Line) (A B : Point) :=
199   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). Decimos que dos puntos están del mismo **lado de una línea** respecto de un punto si los tres puntos están alineados y el segmento que los une no contiene a dicho punto.

```

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

```

src/order_geometry/basic.lean

5.2.2. Resultados

Los teoremas que se pueden deducir de este grupo de axiomas son considerablemente más complicados de demostrar que los del primer grupo. En particular son pruebas que tienen detalles técnicos difíciles de formalizar. En un artículo [8], Meikle y Fleuriot han formalizado resultados de esta sección utilizando el asistente de demostración Isabelle/Isar. En este trabajo exponen que la demostración del *Teorema 3* del libro de Hilbert [7] incluye pasos que se apoyan en intuiciones proporcionadas por un dibujo y no están completamente justificados desde el punto de vista lógico formal.

El teorema tiene el siguiente enunciado:

Teorema 3. *Dados dos puntos distintos A y C existe un tercer punto B que se encuentra entre ellos: $A * B * C$.*

```

17 lemma point_between_given {A C : Point} (hAC : A ≠ C) : ∃ B : Point, A * B * C :=

```

src/order_geometry/propositions.lean

Debido a la complejidad de la demostración, explicada en detalle en el artículo [8], y a la falta de tiempo, no hemos formalizado la demostración de este resultado. Pero nos ha resultado particularmente interesante la lectura de este trabajo, que nos ha descubierto una aplicación más de la formalización asistida por computadores: estos métodos formales permiten estudiar resultados publicados en todo su detalle, no sólo verificando su corrección sino además aportando información sobre la forma de trabajar y pensar de grandes matemáticos como Hilbert. Citando al artículo:

Es interesante señalar que la suposición predominante en matemáticas es que las demostraciones de Hilbert parecen menos intuitivas, pero tienen un rigor aumentado. La prueba Isar muestra que el trabajo de Hilbert puede hacerse aún más riguroso con la ayuda de máquinas. Hilbert claramente utilizó un diagrama para ayudar a la intuición geométrica y hacer muchas suposiciones. Esto parece estar en desacuerdo con su afirmación de que no se necesitaba intuición geométrica para derivar los teoremas presentados en el Grundlagen.

El resultado de esta sección que sí hemos formalizado se refiere a la relación de *estar del mismo lado del plano respecto de una línea*.

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

La librería *mathlib* contiene definiciones sobre relaciones de equivalencia que podemos utilizar: las definiciones de reflexividad, simetría y transitividad. Por tanto se tienen tres resultados referentes a estas tres propiedades. Hemos incluido las demostraciones de las dos primeras propiedades, pero no la de la transitividad⁸, que se puede consultar en el repositorio.

```
25 lemma same_side_line_refl (l : Line) :
26   reflexive (@same_side_line Point Line og l) :=
27   begin
28     intro P,
29     left,
30     refl,
31   end
```

src/order_geometry/propositions.lean

```
34 lemma same_side_line_symm (l : Line) :
35   symmetric (@same_side_line Point Line og l) :=
36   begin
37     intros P Q h,
38     cases h with h1 h2,
39     { left, rw h1 },
40     { cases h2 with h h2,
41       right,
42       use h.symm,
43       rw seg_intersect_line at h2 ⊢,
44       push_neg at h2 ⊢,
45       intros A hA,
46       apply h2 A,
47       rw seg_mem_symm,
48       exact hA,
49     },
50   end
```

src/order_geometry/propositions.lean

```
59 lemma same_side_line_trans (l : Line) :
60   transitive (@same_side_line Point Line og l) :=
```

src/order_geometry/propositions.lean

Con esto, en *mathlib*, las relaciones de equivalencia se definen como estructuras que contienen las demostraciones de estas tres propiedades:

```
201 theorem same_side_line_equiv (l : Line) :
202   equivalence (@same_side_line Point Line og l) :=
```

⁸No hemos podido concluir la formalización de esta demostración debido a su complejidad y longitud. En el código está incluido el esquema general de la demostración, con algún *sorry* pendiente de completar. Se puede consultar el código en el repositorio (ver apéndice A).

```

203   (same_side_line_refl Point l,
204     same_side_line_symm Point l,
205     same_side_line_trans Point l)

```

src/order_geometry/propositions.lean

5.3. Geometría de congruencia

El tercer grupo de axiomas establece propiedades de las relaciones de *congruencia entre segmentos* y *congruencia entre ángulos*. La relación de *congruencia entre segmentos* es una relación binaria entre *segmentos*. La relación de *congruencia entre ángulos* es una relación binaria entre *ángulos*. Usaremos el mismo símbolo \cong para denotar ambas relaciones y por el contexto quedará claro a cual nos referimos.

De la misma forma que en la geometría del orden, queremos considerar también los axiomas anteriores, por lo que extenderemos la clase *order_geometry*.

```

18 class congruence_geometry (Point Line : Type*)
19   extends order_geometry Point Line :=
20   (scong : Seg Point → Seg Point → Prop)
21   (acong : Ang Point Line → Ang Point Line → Prop)

```

src/congruence_geometry/basic.lean

5.3.1. Congruencia de 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 $\overline{AB} \cong \overline{CD}$.*

```

22 (C1 (s : Seg Point) (A B : Point) (hAB : A ≠ B) :
23   ∃! C : Point, ∃ hBC : B ≠ C, ∃ hAC : A ≠ C,
24     (same_side_point Line A B C hBC)
25     ∧ (scong s (Seg.mk hAC))
26 )

```

src/congruence_geometry/basic.lean

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.*

```

27 (C21 {s1 s2 s3 : Seg Point} (h1 : scong s1 s2) (h2 : scong s1 s3) :
28   scong s2 s3)
29 (C22 (s : Seg Point) : scong s s)

```

src/congruence_geometry/basic.lean

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 $\overline{AC} \cong \overline{DF}$.*

```

30 (C3 {A B C D E F : Point}
31   (hABC : neq3 A B C ∧ between A B C)
32   (hDEF : neq3 D E F ∧ between D E F)
33   (h1 : scong (Seg.mk hABC.1.1) (Seg.mk hDEF.1.1))
34   (h2 : scong (Seg.mk hABC.1.2.2) (Seg.mk hDEF.1.2.2)) :
35   scong (Seg.mk hABC.1.2.1) (Seg.mk hDEF.1.2.1)
36 )

```

src/congruence_geometry/basic.lean

5.3.2. Congruencia de ángulos

Axioma C4. *Dado un ángulo α y punto distintos A y B existe un punto C tal que $\angle CAB \cong \alpha$. Este punto C está univocamente determinado si fijamos un lado de la línea \overline{AB} .*

```

37 (C4 (α : Ang Point Line) (A B Side : Point) (hAB : A ≠ B)
38   (hSide : ¬ Side ~ (line Line hAB).val) :
39   ∃! C : Point, ∃ h : same_side_line (line Line hAB).val Side C,
40   (acong α (Ang.mk_from_points B A C
41     (same_side_line_non_collinear hAB hSide h))))

```

src/congruence_geometry/basic.lean

El punto Side y la hipótesis hSide se han utilizado para fijar un lado de la línea \overline{AB} .

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.*

```

42 (C51 {α β γ : Ang Point Line} (hαβ : acong α β) (hαγ : acong α γ) :
43   acong β γ)
44 (C52 (α : Ang Point Line) : acong α α)

```

src/congruence_geometry/basic.lean

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 $\angle ACB \cong \angle DFE$.*

```

45 (C6 (T1 T2 : Tri Point Line)
46   (h1 : scong (Seg.mk T1.neq.1) (Seg.mk T2.neq.1))
47   (h2 : scong (Seg.mk T1.neq.2.1) (Seg.mk T2.neq.2.1))
48   (h3 : acong (Ang.mk_from_points T1.B T1.A T1.C T1.non_collinear)

```

```

49         (Ang.mk_from_points T2.B T2.A T2.C T2.non_collinear)) :
50     scong (Seg.mk T1.neq.2.2) (Seg.mk T2.neq.2.2)
51   ^ acong
52     (Ang.mk_from_points T1.A T1.B T1.C T1.non_collinear_symm)
53     (Ang.mk_from_points T2.A T2.B T2.C T2.non_collinear_symm)
54   ^ acong
55     (Ang.mk_from_points T1.A T1.C T1.B T1.non_collinear_symm')
56     (Ang.mk_from_points T2.A T2.C T2.B T2.non_collinear_symm')
57 )

```

src/congruence_geometry/basic.lean

5.3.3. Resultados

En esta sección solo hemos formalizado las dos proposiciones que establecen que las relaciones de *congruencia de segmentos* y *congruencia de ángulos* son relaciones de equivalencia. Los enunciados y demostraciones son análogos, por lo que incluimos solamente el caso de los segmentos.

```

61 variables (Point Line : Type) [cg : congruence_geometry Point Line]
62
63 /- La congruencia de segmentos es reflexiva.-/
64 lemma scong_refl : reflexive (cg.scong) :=
65 begin
66   intro seg,
67   exact cg.C22 seg,
68 end
69
70 /- La congruencia de segmentos es simétrica.-/
71 lemma scong_symm : symmetric (cg.scong) :=
72 begin
73   intros s1 s2 h1,
74   exact cg.C21 h1 (scong_refl Point Line s1),
75 end
76
77 /- La congruencia de segmentos es transitiva.-/
78 lemma scong_trans : transitive (cg.scong) :=
79 begin
80   intros s1 s2 s3 h12 h23,
81   exact cg.C21 (scong_symm Point Line h12) h23,
82 end
83
84 /- La congruencia de segmentos es una relación de equivalencia.-/
85 theorem scong_equiv : equivalence (cg.scong) :=
86   ⟨scong_refl Point Line, scong_symm Point Line, scong_trans Point Line⟩

```

src/congruence_geometry/basic.lean

5.4. El axioma de las paralelas

El estudio del quinto postulado de la geometría de los *Elementos* ha sido crucial en la historia de la geometría. Empecemos definiendo el concepto de *líneas paralelas* y enunciando el axioma.

Definición (Líneas paralelas). Se dice que dos líneas l y m son **paralelas** si no tienen ningún punto en común.

Una de las formas equivalentes en las que se puede enunciar el axioma de las paralelas es la siguiente:

```
6 def parallel (Point : Type*) {Line : Type*} [ig : incidence_geometry Point Line]
7   (l m : Line) : Prop := ¬ ∃ P : Point, is_common_point P l m
```

src/parallels_independence.lean

Axioma P. Dada una línea l y un punto A existe una única línea m que pasa por A y es paralela a l .

```
11 def P (Point Line : Type*) [ig : incidence_geometry Point Line] :=
12   ∀ (l : Line) (A : Point), ¬ A ~ l → ∃! m : Line, A ~ m ∧ parallel Point l m
```

src/parallels_independence.lean

La cuestión principal a determinar, que ha sido un problema abierto durante varios siglos, hasta la solución del problema en el siglo diecinueve con el desarrollo de las geometrías no euclídeas, es si el axioma de las paralelas puede deducirse a partir de los otros axiomas. Es decir, ¿es posible prescindir del quinto postulado de *Euclides*?

En lógica moderna llamamos a esta idea *independencia*. Si no existe forma de derivar una fórmula dada de un conjunto de fórmulas, decimos que nuestra fórmula es independiente del resto. Fijados unos axiomas, decir que una proposición es independiente de ellos equivale a decir que incluyendo su negación como axioma obtenemos una teoría consistente, en la que no se pueden demostrar contradicciones. Esta noción, expresada en términos sintácticos, tiene su equivalente semántico. Si fijados unos axiomas podemos construir un modelo en el que se cumple la proposición y otro en el que se cumple su contradicción.

Diversos libros modernos de geometría euclídea axiomática llaman al conjunto de axiomas que hemos presentado hasta ahora planos de Hilbert (los modelos de esta teoría reciben el mismo nombre). En esta teoría se cumplen los axiomas de incidencia, orden y congruencia, pero no se establece el axioma de las paralelas. Con estos términos, decir que el axioma de las paralelas es independiente del resto de axiomas equivale a decir que existen planos de Hilbert en los que se cumple el axioma y otros en los que se cumple su negación. Con el siguiente teorema hemos formalizado esta idea:

```
128 theorem congruence_geometry_parallels_independence (Point Line : Type*) :
129   ¬ (∀ cg : congruence_geometry Point Line,
130     @P Point Line cg.to_incidence_geometry) :=
```

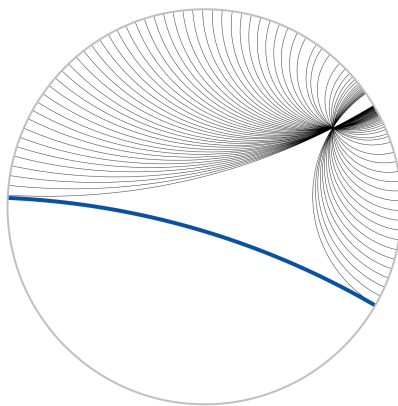
src/parallels_independence.lean

Usando la táctica `push_neg` modificamos la meta inicial a

```
⊢ ∃ cg : congruence_geometry Point Line, ¬ @P Point Line cg.to_incidence_geometry.
```

Una forma de demostrar este resultado es por tanto construir un término de tipo `cg : congruence_geometry Point Line` para el cual no se cumpla el axioma `P`. Para construir dicho término habría que definir una instancia de la clase `congruence_geometry`. Observamos un paralelismo entre las nociones de instancias de clases y modelos de teorías.

Algunos modelos simples de planos euclídeos donde no se cumple el axioma de las paralelas son el *disco de Poincaré* y el *disco de Beltrami*.



En el disco de Poincaré existen infinitas líneas paralelas a una dada que pasan por un punto fijo.

Podríamos considerar estos modelos y formalizar sus construcciones en Lean. Para ello habría que definir los tipos *Line* y *Point*, las distintas relaciones de incidencia, orden y congruencia para estos tipos, demostrar que se cumplen los axiomas de los planos de Hilbert pero no el de las paralelas. Esto se puede hacer precisamente mediante el mecanismo de las instancias de clases.

6. Conclusiones

La lectura del artículo de Meikle y Fleuriot [8] nos ha resultado sumamente interesante. Lo consideramos una prueba más de que sistemas de verificación formal como Lean tienen mucho que aportar a las matemáticas y nos ayudan a conocer cómo razonaban grandes matemáticos.

Encontrando detalles no totalmente justificados en demostraciones escritas hace más de un siglo por Hilbert descubrimos que el objetivo de Hilbert de separar el trabajo deductivo formal de la intuición geométrica no es realizable por completo. Hilbert utilizó razonamientos en los que aparentemente cada paso estaba fundado exclusivamente en el uso de una lógica precisa y en los pasos anteriores. Pero en realidad ciertos pasos que pueden resultar evidentes si pensamos en el dibujo

correspondiente, necesitan ser desarrollados. No se pueden separar totalmente la intuición matemática y su formalización.

Las herramientas de formalización en cierto sentido cumplen el proyecto empezado por Hilbert, mostrando que se puede llevar la formalización hasta su extremo, comprobando automáticamente cada detalle. Pero a su vez muestran sus propias limitaciones. De nada sirve la formalización si no tenemos un contenido matemático que formalizar y una intuición con la que interpretar y aprender de la nueva información que aportan sistemas como Lean.

Hemos aprendido también que no todo el trabajo está en desarrollar código en Lean. A veces no es inmediato cómo formalizar una definición o proposición. La forma más intuitiva y práctica en matemáticas no tiene por qué tener una correspondencia directa con la formalización que tenga más sentido realizar en Lean. Es necesario conocer muy bien las teorías, conceptos y definiciones que se están formalizando a un nivel de abstracción suficientemente amplio como para ser capaces de desarrollar formulaciones equivalentes más convenientes para sus formalizaciones.

Más de una vez, a la hora de formalizar una demostración de una proposición de la geometría euclídea en Lean hemos descubierto que no habíamos entendido por completo la demostración. Lean ayuda enormemente a aislar e identificar las partes de una demostración que no comprendemos. Si no entendemos por completo un paso o una parte de una demostración no sabremos formalizarla, e incluimos un **sorry** en el código. A partir de aquí entramos en un proceso iterativo en el que volvemos al texto de la prueba para enfocarnos en los detalles restantes, y poco a poco vamos eliminando los **sorrys**.

En las formalizaciones realizadas en este trabajo no hemos pretendido alcanzar una síntesis y claridad óptima en las demostraciones, sino conseguir que el sistema las acepte. Muchas de las pruebas pueden ser acortadas, por ejemplo extrayendo partes a lemas adicionales para ser reutilizadas. Seguramente no hayamos utilizado siempre las tácticas más adecuadas, habiendo otras posibilidades más elegantes y claras de leer.

Las principales dificultades han surgido a la hora de intentar formalizar ejemplos concretos, como modelos de los distintos grupos de axiomas que se consideran. Por ejemplo, no he conseguido terminar de formalizar el modelo más simple de una geometría de incidencia en el que no se cumple el axioma de las paralelas.

En parte atribuimos estas dificultades a la falta de una documentación más completa y accesible de la librería *mathlib*, en la que se explique cómo están implementados ciertos objetos matemáticos en ella y qué utilidades se proporcionan para trabajar con ellos.

Al no haber estudiado en detalle el funcionamiento del lenguaje Lean y la estructura de la librería *mathlib*, nos hemos encontrado más de una vez con comportamientos y errores, que no hemos sabido interpretar. De nuevo, los problemas más recurrentes han surgido de las dificultades encontradas al intentar entender el código disponible en la librería *mathlib*.

Hemos digitalizado solo una pequeña parte de la geometría de Hilbert. Invitamos a los lectores interesados en este trabajo a participar en él, añadiendo los axiomas y definiciones que faltan, demostrando resultados o proponiendo cambios. Para ello se pueden crear *tíquets* o *merge requests* en el repositorio de *github*, enlazado en el apéndice [A](#).

A. Repositorio y organización del código

El código desarrollado en este trabajo se encuentra en un *repositorio git*, alojado en el siguiente enlace: <https://github.com/haztecaso/euclidean-geometry-lean>.

En este repositorio se encuentran todas las demostraciones que se han desarrollado pero no han sido incluidas en este documento debido a su extensión.

La estructura de los ficheros del repositorio es la siguiente:

```
├── memoria
│   ├── memoria.pdf
│   └── presentacion.pdf
├── src
│   ├── basic_defs.lean
│   ├── incidence_geometry
│   │   ├── basic.lean
│   │   └── propositions.lean
│   ├── order_geometry
│   │   ├── basic.lean
│   │   └── propositions.lean
│   ├── congruence_geometry
│   │   └── basic.lean
│   ├── parallels_independence.lean
│   └── examples.lean
└── readme.md
```

En el fichero `readme.md`, que se puede leer cómodamente en el enlace del repositorio, se pueden consultar las instrucciones de ejecución del código, junto a una breve descripción del contenido de cada fichero.

B. Entorno de desarrollo

Visual Studio Code

El entorno recomendado⁹ para desarrollar código Lean es el editor *Visual Studio Code*¹⁰. En este editor, una vez instalado el plugin que da soporte a Lean, tenemos acceso a una serie de herramientas que facilitan la lectura y desarrollo de código en Lean. A continuación exponemos alguna de estas herramientas.

⁹Lean también tiene soporte mediante plugins para los editores *emacs* y *vim*.

¹⁰Aconsejamos utilizar la distribución de *Visual Studio Code* de software libre, [VSCodium](#).

Consulta de tipos

Moviendo el cursor por encima de definiciones o tácticas podemos obtener información adicional sobre un término, tipo o táctica, accediendo a su tipo, su definición o la documentación correspondiente.

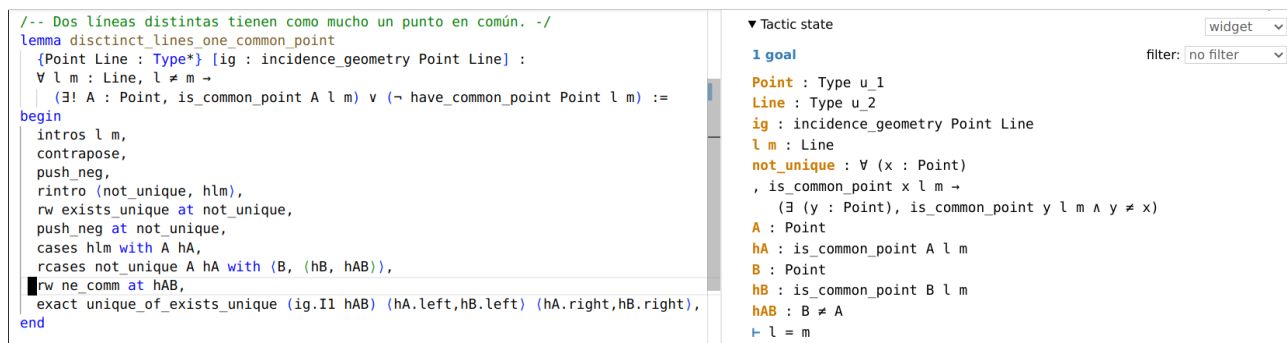
Símbolos matemáticos

Como se ha mencionado anteriormente en este documento, en Lean se pueden utilizar símbolos unicode para expresar tipos útiles en la formalización de matemáticas y obtener así un código más cercano a las notaciones lógicas 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 \rightarrow , `\lambda` por λ y `\N` por \mathbb{N} . Si mantenemos el cursor encima de un símbolo, un panel informativo nos indicará el código que podemos utilizar para escribirlo.

Panel de estado táctico

La herramienta más importante de este entorno es el panel lateral desde el cual podemos consultar el estado táctico. En la siguiente captura de pantalla se ve cómo, situando el cursor en un paso de una demostración, el panel lateral proporciona toda la información contextual del estado táctico correspondiente.



Captura de pantalla del entorno de desarrollo en el editor *Visual Studio Code*.

Durante la formalización o lectura de una demostración en Lean estamos continuamente moviéndonos a través del código y observando de qué forma varía el estado táctico.

Referencias

- [1] Kevin Buzzard. *Formalising Mathematics — Formalising Mathematics 0.1 Documentation*. URL: <https://www.ma.imperial.ac.uk/~buzzard/xena/formalising-mathematics-2023/> (visitado 31-08-2023).
- [2] Jeremy Avigad y Floris van Doorn. «The Lean Theorem Prover and Homotopy Type Theory». En: ().
- [3] María Inés de Frutos. «Formalización de Matemáticas En Lean». Curso de Doctorado En Investigación Matemática (UCM). URL: https://www.ucm.es/doctorado/doctorado_inv_mat/noticias/57978 (visitado 05-09-2023).
- [4] «Homotopy Type Theory: Univalent Foundations of Mathematics». En: ().
- [5] 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](https://doi.org/10.1007/978-0-387-22676-7). URL: <http://link.springer.com/10.1007/978-0-387-22676-7> (visitado 27-03-2023).
- [6] 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.
- [7] David Hilbert. *Foundations of Geometry*. Reprint edition. La Salle, Illinois: Open Court Publishing Company, 1950. URL: <https://math.berkeley.edu/~wodzicki/160/Hilbert.pdf>.
- [8] Laura I. Meikle y Jacques D. Fleuriot. «Formalizing Hilbert’s Grundlagen in Isabelle/Isar». En: *Theorem Proving in Higher Order Logics*. Ed. por David Basin y Burkhart Wolff. Red. de Gerhard Goos, Juris Hartmanis y Jan van Leeuwen. Vol. 2758. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, págs. 319-334. ISBN: 978-3-540-40664-8 978-3-540-45130-3. DOI: [10.1007/10930755_21](https://doi.org/10.1007/10930755_21). URL: http://link.springer.com/10.1007/10930755_21 (visitado 27-03-2023).
- [9] Christophe Dehlinger, Jean-François Dufourd y Pascal Schreck. «Higher-Order Intuitionistic Formalization and Proofs in Hilbert’s Elementary Geometry». En: *Automated Deduction in Geometry: Third International Workshop, ADG 2000 Zurich, Switzerland, September 25–27, 2000 Revised Papers 3*. Springer. 2001, págs. 306-323.
- [10] María Inés de Frutos. «¿Por Qué Formalizar Matemáticas?» (Facultad de Matemáticas - Universidad Complutense de Madrid). 14 de mar. de 2023. URL: <https://www.youtube.com/watch?v=4QXWyadT03I> (visitado 03-09-2023).
- [11] Peter Scholze. «Liquid Tensor Experiment». En: *Experimental Mathematics* 31.2 (3 de abr. de 2022), págs. 349-354. ISSN: 1058-6458, 1944-950X. DOI: [10.1080/10586458.2021.1926016](https://doi.org/10.1080/10586458.2021.1926016).
- [12] Peter Scholze. *Blueprint for the Liquid Tensor Experiment*. URL: <https://leanprover-community.github.io/liquid/> (visitado 31-08-2023).

- [13] Thomas Hales. *Formal Abstracts*. Formal Abstracts. URL: <https://formalabstracts.github.io/> (visitado 31-08-2023).
- [14] Stanislas Polu, Jesse Michael Han e Ilya Sutskever. *Solving (Some) Formal Math Olympiad Problems*. URL: <https://openai.com/research/formal-math> (visitado 31-08-2023).

Agradecimientos

Al profesor Jorge Carmona Ruber, por su constante ayuda y buenas conversaciones.
Al gran equipo de personas de BioCoRe, sin los cuales este último año de carrera y redacción de este trabajo hubiera sido mucho más duro.

A mis padres, Florencia y Leandro, y a mi familia, por su incansable apoyo e interés.
A mis amigos y compañeros, con los que hemos compartido gracias y desgracias de estos intensos años de carrera.