1. Formalizando resultados básicos

En esta sección se presentan algunos axiomas y resultados elementales de la axiomática de Hilbert, comparando los enunciados y demostraciones expresados de forma natural (**Pendiente:** ¿Cómo expresar esto bien?) con sus correspondientes formalizaciones en Lean.

1.1. Los axiomas de incidencia

Pendiente: Explicar cómo funcionan las clases, comparar los axiomas de hilbert expresados en lenguaje natural con sus formalizaciones. Diferencias entre parámetros implícitos y explícitos.

Axioma I1. *Para cada par de puntos distintos A, 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 $AB \neq BC$.*

```
class incidence_geometry (Point Line : Type*) :=
  (lies_on : Point → Line → Prop)
  (infix ` ~ ` : 50 := lies_on)
  (I1 {A B : Point} (h : A ≠ B): ∃! l : Line, A ~ l ∧ B ~ l)
  (I2 (l : Line) : ∃ A B : Point, A ≠ B ∧ A ~ l ∧ B ~ l)
  (I3 : ∃ A B C : Point, different3 A B C ∧ ¬ ∃ l : Line,  A ~ l ∧ B ~ l ∧ C ~ l)
```

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.*

Su correspondiente formalización en Lean es:

```
lemma disctinct_lines_one_common_point
  {Point Line : Type*} [ig : incidence_geometry Point Line] :
  ∀ l m : Line, l ≠ m →
    (∃! A : Point, is_common_point A l m) ∨ (¬ have_common_point Point l m) :=
begin
  sorry
end
```

Se puede observar que, gracias al uso de caracteres unicode, la formalización en Lean es muy fácil de leer y cercana a la forma de escribir matemáticas a la que estamos acostumbrados.

Pendiente: ¿Incidir sobre la diferencia entre parámetros explícitos e implícitos?

Observemos que en el enunciado se están utilizando algunas definiciones que se han declarado previamente:

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

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

Pendiente: Aquí si que sería conveniente explicar por qué algunos parámetros son implícitos y otros explícitos.

La demostración, como la presenta el libro de Hartshorne (**Pendiente:** citar), es como sigue:

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 puede interpretarse como una demostración por 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 intentar implementar esta idea en Lean nos damos cuenta de que hay bastantes detalles que necesitamos tener en cuenta.

Esta es la formalización completa del resultado en Lean, incluyendo el enunciado y su demostración:

```
lemma disctinct_lines_one_common_point
  {Point Line : Type*} [ig : incidence_geometry Point Line] :
  ∀ l m : Line, l ≠ m →
    (∃! A : Point, is_common_point A l m) ∨ (¬ have_common_point Point l m) :=
begin
  intros l m,
  contrapose,
  push_neg,
  rintro ⟨not_unique, hlm⟩,
  rw exists_unique at not_unique,
  push_neg at not_unique,
  cases hlm with A hA,
  rcases not_unique A hA with ⟨B, ⟨hB, hAB⟩⟩,
  rw ne_comm at hAB,
  exact unique_of_exists_unique (ig.I1 hAB) ⟨hA.left,hB.left⟩ ⟨hA.right,hB.right⟩,
end
```

Pendiente: Explicar qué es una demostración en Lean. Que es el estado táctico y la meta, qué hacen las tácticas.

Analicemos detalladamente cada uno de los pasos:

- 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.

- `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"

- `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`

- `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.

- `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`.

- `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
  ```

- `push_neg at not_unique,`

- `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
  ```

- `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
  ```

- `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`.

- `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.