Proof Assistants: Lean

David Ávila Jiménez y Manuel González González

Enero de 2022

Índice

1. Introducción

1.1. ¿Qué es un Proof Assistant?

Los Asistentes de Demostraciones son sistemas informáticos que permiten al usuario demostrar y definir conceptos matemáticos. Así, un usuario puede realizar una teoría matemática, definir sus propiedades y aplicar un razonamiento lógico con ellos. Estos asistentes son usados principalmente por especialistas para realizar teorías matemática y demostrar teoremas.
La idea de los Proof Assistants apareció alrededor de los años 70. Ejemplos de teoremas que se han podido demostrar con estos asistentes son: el teorema de los 4 colores, en el lenguaje Coq; el teorema de los números primos realizado en Isabelle, entre otros. Todos ellos creando controversia en la comunidad matemática. De entre los Proof Assistants disponibles, en este documento se trabajará con el asistente Lean.

1.2. Lean

Lean es un proyecto, escrito en C++, de Microsoft Research Redmond lanzado e 2013 por Leonardo de Moura. Se ha publicado bajo la licencia de Apache 2.0 (una licencia de código abierto bajo la que otros usuarios pueden usar y ampliar el código).

El proyecto está en desarrollo y se encuentra en la versión 4, implementada en el propio lenguaje Lean, dejando C++ solo para el kernel. Se puede ver en los sitios oficiales la guía de ruta que tienen pensada para implementar futuras características. En este trabajo nos centraremos en las cosa que ya son posibles de realizar.

Hay por ahora dos formas de usar Lean:

1. Por medio de la web en una versión hecha en Javascript. Para ello se descarga una librería de definiciones y un editor de texto en el navegador desde el que se ejecuta. (Esta se encuentra en una versión anterior y si intentas ejecutar características de Lean 4 da fallos).

2. Instalarlo en el ordenador de forma nativa. Esta forma es más rápida y flexible que la versión web; tiene soporte en Visual Studio Code y Emacs, lo cual facilita escribir y hacer debug de demostraciones. Aunque todo sea dicho, la instalación es tediosa (se aceptan propuestas en su git para mejorar el instalador).

Lean es tanto una herramienta para demostración como un lenguaje de programación. Como lenguaje se encuentra en el paradigma funcional. Además, es un metalenguaje, ya que puede hablar tanto sobre las propiedades de otros lenguajes y como de las suyas mismas. Como ya se ha comentado antes se encuentra en la versión 4 que amplia y modifica mucho el lenguaje respecto a la versión

3. En esta práctica se trabajará sobre la versión 3 debido a que los conceptos que consideramos necesarios para la explicación ya se encuentran en la versión 3. Además de que tanto la página web como la versión que hemos conseguido instalar es esta. Lo que implica que las características de Lean 4 no pueden ser ejecutadas.
Lean se encuentra todavía en fase desarrollo y no está pensado todavía para hacer proyectos reales dado que cada actualización realiza cambios muy bruscos. De todos modos en la bibliografía se dejarán los enlaces a los dos manuales que se encuentran en la página de Lean.

2. Teoría Dependiente de Tipos

La teoría dependiente de tipos es una forma muy potente de expresar afirmaciones matemáticas complejas. Lean se basa en una versión de la teoría de tipos dependientes conocida como el Cálculo de Construcciones, con una jerarquía contable de universos no acumulativos y tipos inductivos.
Partimos de que la teoría de conjuntos tiene una ontología simple pero atractiva. Todo puede basarse en conjuntos y esto es lo que consigue Lean con la teoría de tipos ya que cada expresión tiene un tipo asociado (conjunto al que pertenece). Ejemplo sería la expresión $\boxed{\mathbf{n + 0}}$ que puede denotar un número natural, también se pueden referir a cadenas (string), booleanos (bool).
La teoría de tipos es muy poderosa ya que nos permite construir nuevos tipos a partir de otros. Además Lean tiene una jerarquía infinita de tipos (Type 0, Type 1,...) , en donde un tipo contiene a otros tipos. Estas variables Type se usan cuando queremos construir tipos que tengan una mayor generalización. A continuación mostramos un ejemplo de los tipos de Lean.
Tras realizar esto con el comando **check** podemos pedir a Lean que nos muestre el tipo al que pertenece.

Disculpamos la ausencia de tildes en los bloques de código, pero se representan mal.

```
/- Los comentarios se denotan mediante -- Texto (afecta solo una
    linea) o mediante /- Texto -/ (puede afectar a mas de una
    linea),
son partes del codigo que Lean no interpreta. -/

/- Definicion de constantes -/

def m : nat := 1        -- m es numero natural
def n : Type := nat     -- n es un Tipo 1
def b1 : bool := true   -- b1 es un booleano
def b2 : bool := false  -- b2 es un booleano

/- Con el comando def introducimos una nueva constante en el
    entorno de trabajo, siendo indicado su tipo. Como se basa en
    la teoria de tipos tenemos infinitos de estos, puediendo
    utilizar desde nat (Numeros naturales) o bool (tipos
    booleanos) hasta otro tipos como son Type, Type 1, Type 2 y
    que pueden englobar unos a otros. Tambien permite introducir
    funciones o proposiciones entre otras cosas. -/
```

```
/- Comprobacion de tipos -/

#check m                -- Nat
#check n                -- Type 1
#check b1               -- Bool
#check Type 1           -- Type 2
#check true             -- Boolean "true"

/- El comando check nos muestra a que tipo pertenece una variable
    , constante o defincin -/

/- Evaluacion -/

#eval 5 * 4 -- 20
#eval m + 2 -- 3
#eval b1 && b2 -- false

 /- El comando #eval nos muestra el resultado por pantalla de la
    expresion a evaluar. -/
```

```
/- Funciones -/

def double (n : ℕ) : ℕ := n + n
#eval double 1 -- 2

def thrice : ℕ → ℕ := λ x, x + x + x
#eval thrice 2 -- 6

/- El comando #reduce dice a Lean que evalue una expresion por
    reduccion a su forma normal, es decir, realizando todas las
    reducciones computacionales  -/

constants x y : nat
constant b : bool
-- Reduciendo tuplas
#reduce (x, y).1 -- x
#reduce (x, y).2 -- y
-- Reduciendo expresiones booleanas
#reduce tt && ff -- ff
#reduce ff && b -- ff
-- Reduciendo expresiones aritmeticas
#reduce y + 0 -- n
#reduce y + 2 -- y.succ.succ
#reduce 2 + 3 -- 5
```

Lean también incluye librerías para trabajar con demostraciones en ámbitos específicos, por ejemplo init.data.int incorpora algunas demostraciones automáticas. Véase:

```
import init.data.int.basic

variables a b c d : ℤ

example : a + 0 = a := int.add_zero a
example : 0 + a = a := int.zero_add a
example : a + b = b + a := int.add_comm a b
-- Si en algun momento necesitamos dejar por hacer una
    demostracion, podemos usar sorry
example : (a + b) * c = a * c + b * c := sorry
-- De esta forma no queda demostrado, pero nos permite seguir
    realizando la demostracion por otro lado sin que de error. Es
     equivalente al pass en Python o undefined en Haskell
```

3. Expresiones matemáticas en Lean

Como hemos podido observar en la sección anterior, Lean nos permite escribir de forma cómoda simbología matemática de forma similar a LaTeX. Algunos ejemplos son:

```
∅ \emptyset     ∈ \in               ∉ \not in
∋ \ni           ⊆ \subset           ⊆ \subseteq
⊇ \supset       ⊇ \supseteq         ⊄ \nsubset
⊈ \nsubseteq    ∪ \cup o \u         ∩ \cap o \i
∀ \forall       ¬ \n o \neg         ∨ \or
∧ \and          → \p, \to o \r      ↔ \iff
```

4. Demostración de ejemplo de un tipo inductivo

Vamos a ver un ejemplo de demostración de una propiedad sobre un tipo inductivo.

```
namespace hidden
/- namespace indica un bloque independiente dentro del codigo que
     nos permite reusar el nombre de elementos sin que haya error
     de renombramiento, en este caso para definir nat nos hace
    falta porque dicha estructura ya se encuentra implementada en
     Lean -/

inductive nat : Type
| zero : nat
| succ : nat → nat

end hidden

open nat

-- Aqui realizamos una implementacion con el tipo definido nat
def two_pow : hidden.nat → ℕ
| hidden.nat.zero        := 1
| (hidden.nat.succ n) := 2 * two_pow n

-- Demostramos por induccion

example (n : hidden.nat) : two_pow hidden.nat.zero = 1 := rfl
-- rfl o Eq.refl es el cumplmiento de la propiedad reflexiva
example (n : hidden.nat) : two_pow (hidden.nat.succ n) = 2 *
    two_pow n := rfl
```

```
#eval two_pow (hidden.nat.succ hidden.nat.zero) -- 2

-- Y aqui lo hacemos con el ℕ propio de Lean

def two_powN : ℕ → ℕ
| 0        := 1
| (succ n) := 2 * two_powN n

#eval two_powN 2 -- 4

example (n : ℕ) : two_powN 0 = 1 := rfl
example (n : ℕ) : two_powN (n + 1) = 2 * two_powN n := rfl
```

5. Proposiciones

Una vez hablado el sistema de tipos que utiliza Lean, vamos a tratar un tipo en concreto muy útil e importante. Lean incorpora el tipo Prop que se usa para declarar fórmulas proposicionales de la lógica. Como ocurre en la lógica, se definen los elementos mínimos que se entienden por fórmula y las reglas con las que se inducen nuevas. De esta forma se pueden generar nuevas fórmulas proposicionales de forma inductiva.

Podemos ver en el ejemplo siguiente la definición de las reglas **and**, **or**, **not** e **implies**; y la declaración de literales proposicionales.

```
axioms p q r : Prop
#check and p q  -- Prop
#check or (and q r) p  -- Prop
#check p ∧ r  -- Prop
#check r → (q ∨ ¬p)  -- Prop
```

Para trabajar con las proposiciones Lean ofrece **theorem** (y **lemma**), que es prácticamente un definition con la diferencia de que con theorem, el parser no desenrrolla dicha definición, pues no le hace falta saber los detalles, solo la exactitud de una demostración. Esto permite a Lean hacer comprobar demostraciones en paralelo.

Se puede declarar un theorem de varias maneras.
Podemos usar **lamda-abstracciones** o **asunciones** (assume).

```
-- Lambda-abstraccion
variables p q : Prop
theorem t1 : p → q → p := λ hp : p, λ hq : q, hp

-- Asunciones
variables p q : Prop
theorem t1 : p → q → p :=
  assume hp : p,
  assume hq : q,
  hp
```

Las abstracciones lambda **hp : p** y **hq : q** se pueden ver como suposiciones que se hacen temporalmente en la prueba de t1. Además si **p** y **q** han sido declaradas como variables, Lean las generalizará automaticamente. Para theorem se pueden utilizar expresiones lógicas que nos permite introducir y eliminar cada conector lógico con el fin de realizar comprobación de demostraciones. A continuación mostramos como introducir y eliminar las expresiones lógicas.

Conjunción

```
variable (p q : Prop)
example (hp : p) (hq : q) : p ∧ q := And.intro hp hq
```

La expresión **And.intro hp hq** crea una prueba de **p → q → p ∧ q** que sirve para introducir la regla de **conjunción**. También se puede eliminar la regla con la expresión **And.left h** o con **And.right h** siendo pruebas por separado de **p** y **q**, respectivamente.

```
variable (p q : Prop)
example (h : p ∧ q) : p := And.left h
example (h : p ∧ q) : q := And.right h
```

Disyunción

```
variable (p q : Prop)

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

En la disyunción la expresión **And.intro_left q hp** crea una prueba de **p ∨ q** a partir de **hp : p**. Lo mismo ocurre para la expresión **And.intro_left p hq**. Para la eliminar se debe realizar una prueba por casos primero comprobando que se cumple por izquierda y después por derecha.

```
variables p q : Prop
```

```
-- BEGIN
example (h : p ∨ q) : q ∨ p :=
or.elim h
  (assume hp : p,
    show q ∨ p, from or.intro_right q hp)
  (assume hq : q,
    show q ∨ p, from or.intro_left p hq)
-- END
```

Negación y falsedad

La negación ¬p se puede definir de dos formas diferentes mediante la expresión **p→False**, por lo que obtenemos ¬p por derivación de una contradicción de p. Mediante la expresión **hnp : hp** resulta una prueba de false de **hp: p** y **hnp: ¬p**.

```
variables p q : Prop
example (hpq : p → q) (hnq : ¬q) : ¬p :=
assume hp : p,
show false, from hnq (hpq hp)
```

El conectivo False tiene una regla de eliminación, **False.elim** que expresa que cualquier cosa se sigue de una contradicción.

```
variables p q : Prop
example (hp : p) (hnp : ¬p) : q := false.elim (hnp hp)
```

Equivalencia lógica

Para introducir la equivalencia lógica utilizamos la expresión **iff.intro h1 h2** que nos produce una prueba de **p ↔ q** de **h1 : p → q** y **h2 : q → p**. Mientras que para eliminar utilizamos la expresión **iff.elim_left h** produce una prueba **p → q de h : p ↔ q**. De la misma forma, **iff.elim_right h** produce una prueba **q → p de h : p ↔ q**.

```
variables p q : Prop
theorem and_swap : p ∧ q ↔ q ∧ p :=
iff.intro
  (assume h : p ∧ q,
    show q ∧ p, from and.intro (and.right h) (and.left h))
  (assume h : q ∧ p,
    show p ∧ q, from and.intro (and.right h) (and.left h))

#check and_swap p q    -- p ∧ q ↔ q ∧ p
```

La introducción y eliminación de las reglas clásicas nos va a permitir demostrar ejemplos de validez proposicional y que va a ser explicado en el siguiente ejemplo. En este ejemplo mostramos otra forma de declarar theorem, mediante

el comando example, con el cual no se nombra ni se almacena en el contexto, simplemente comprueba que el término dado tiene el tipo indicado.

```
-- Tactic mode
example (p q r : Prop) : ((p ∨ q) → r) ↔ ((p → r) ∧ (q → r))
    :=
begin -- Empezamos el modo tactic
  split, -- Como tenemos que demostrar en ambos sentidos por
         -- la doble implicacion, usamos split para dividir la
    demostracion en dos
  -- Empezamos a demostrar por la parte izquierda
  { intro h, -- Con 'intro' creamos una hipotesis asumiendo
    cierta la premisa de la implicacion
    split, -- Como tenemos que de lo anterior se infiere (p → r)
     ∧ (q → r),
           -- volvemos a dividir la demostracion
    { intro hp, -- Tenemos que demostrar p → r ías que asumimos
    p cierta
      apply h, -- Aplicando la hipotesis nos queda demostrar que
    p ∨ q sea cierto
      left, -- Como hemos asumido p cierta aplicamos left que
    coge
            -- el primer constructor de un tipo inductivo cuando
    hay dos
      assumption}, -- Y aplicamos este tactics que busca en el
    entorno una hipotesis equivalente a la meta
                 -- Si encuentra una, la usa para demostrar la
    meta; si no, falla
    { intro hq,
      apply h,
      right,
      assumption}},
  -- Y continuamos por la parte derecha
  { intro h,
    cases h with hpr hqr, -- separamos la hipotesis en dos casos,
     que se cumpla  p → r o que se cumpla q → r
    intro hpq, -- Y suponemos que p ∨ q es cierto
    cases hpq with hp hq, -- Con ello tenemos otros dos casos:
    que p sea cierto o que lo sea q
    { apply hpr, -- Aplicamos la hipotesis de que p → r
      assumption},
    { apply hqr,
      assumption}}
end

-- Term mode
```

```
example (p q r : Prop) : ((p ∨ q) → r) ↔ ((p → r) ∧ (q → r))
    :=
⟨ λ h, ⟨ λ hp, h $ or.inl hp, λ hq, h $ or.inr hq⟩, λ ⟨ hpr, hqr⟩
    hpq, hpq.elim hpr hqr⟩
```

6. Bibliografía

- H GEUVERS "Proof assistants: History, ideas and future"
- Lean
- Manual de Lean3
- Ejemplo de demostración
- Editor web (versión Lean 3)