Proof Assistants

Lean

David Ávila Jiménez y Manuel González González

Escuela Técnica Superior De Ingeniería Informática
Universidad de Málaga

Enero de 2022

Índice

Introducción: Proof Assistants y Lean

¿Qué es un Proof Assistant?

Un proof assistant es un sistema informático que permite al usuario definir y demostrar conceptos matemáticos.

Existen dirvesos proof assistants como pueden ser Coq, Isabelle o Lean.

Introducción: Proof Assistants y Lean

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

Es una herramienta tanto para ayudarte a realizar demostraciones y como un lenguaje de programación (programación funcional). Además, es un metalenguaje pues puede demostrar propiedades de otros lenguajes y de sí mismo.

Se encuentra en fase desarrollo y no se recomienda para proyectos serios.

Introducción: Proof Assistants y Lean

Lean3 vs Lean4

- Lean3 es la versión disponible en el entorno web de pruebas y es el que se puede instalar con mayor facilidad.
- Lean4 es una versión muy mejorada y ampliada. Pero cambia cosas básicas de la sintaxis de Lean3, por lo que hay que tener cuidado al consultar los manuales.

¿Cuál usamos?

Para los propósitos de esta presentación y por imposibilidad de instalar Lean4, nos centraremos en Lean3

Teoría Dependiente de Tipos

Teoría de tipos

Permite expresar afirmaciones matemáticas complejas. Lean se basa en una versión de esta teoría conocida como el cálculo de construcciones, con una jerarquía contable de universos no acumulativos y tipos inductivos.

Tipos en Lean

La expresión **n + 0** se nota como un numero natural, **tt** como un booleano. Además contiene una jerarquía infinita de tipos **Type 0**, **Type 1**, etc, que se contienen unas a otras y permite una mayor generalización.

Teoría Dependiente de Tipos

Declaración

```
/- 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
```

Check

```
/- Comprobacion de tipos -/

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

Teoría Dependiente de Tipos

Eval

```
/- Evaluacion -/

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

Funciones

```
/- 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
```

Reduce

```
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
```

Autodemostración

```
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
```

Expresiones matemáticas

¿Familiarizado con LaTeX?

$\emptyset$ \emptyset	$\in$ \in	$\notin$ \not in
$\ni$ \ni	$\subset$ \subset	$\subseteq$ \subseteq
$\supset$ \supset	$\supseteq$ \supseteq	$\not\subset$ \nsubset
$\not\subseteq$ \nsubseteq	$\cup$ \cup o \u	$\cap$ \cap o \i
$\forall$ \forall	$\neg$ \n o \neg	$\vee$ \or
$\wedge$ \and	$\to$ \p, \to o \r	$\leftrightarrow$ \iff

Ejemplo de Demostración

```
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
```

Ejemplo de Demostración

```
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
```

Ejemplo de Demostración

```
#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
```

Proposiciones

Tipo Prop

Lean incorpora el tipo Prop que se usa para declarar fórmulas proposicionales de la lógica.

Como ocurre en lógica se definen los elementos mínimos que se entienden por fórmula y las reglas con que se inducen nuevas fórmulas 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
```

Proposiciones

Theorem y Lemma

Antes hemos usado **example** para demostrar una propiedad matemática. Pues bien, eso es un caso anónimo de **theorem** o **lemma**. Podemos querer guardar un teorema como si de una *función* se tratara para usarlo luego en otro teorema.

Hay varias formas de representar teoremas, al igual que pasaba con las funciones.

Proposiciones

```
-- 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
```

Bibliografía

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