¿Cómo funciona un asistente de demostración?

If you can't explain your mathematics to a machine it is an illusion to think you can explain it to a student. –N. G. de Bruijn

¿Qué es un asistente de demostración?

¿Qué **no** es un asistente de demostración?

- ✗ *Computer algebra system* — Mathematica, Matlab, Sage.
- ✗ *Automated theorem prover* — E, Otter, CVC, Yices.
- ✗ Lenguaje de programación funcional — ¿...o sí?

¿Qué es un asistente de demostración?

¿Qué **no** es un asistente de demostración?

- ✗ *Computer algebra system* — Mathematica, Matlab, Sage.
- ✗ *Automated theorem prover* — E, Otter, CVC, Yices.
- ✗ Lenguaje de programación funcional — ¿...o sí?

¿Qué **sí** es?

- ✓ Herramienta que permite escribir y verificar formalmente:
  - Definiciones.
  - Proposiciones.
  - Algoritmos.
  - Demostraciones.
- ✓ Ayudar en este proceso:
  - Desenrollar las definiciones.
  - Mantener registro de las hipótesis y objetivos.
  - Aplicar reglas de lógica o reescritura.
  - Automatizar (algunas) demostraciones.
  - Extraer programas correctos.

Cuestiones interesantes (a esquivar hoy)

- [?] ¿Queremos formalizar la matemática? ¿Quiénes? ¿Para qué?
- [?] ¿Qué relación costo-beneficio tiene?
- [?] ¿En qué hay que confiar para tener la convicción de que una demostración aceptada por un asistente es "correcta"?
- [?] ¿Cuán razonable es confiar en eso?
- [?] ¿Cuán legible es, comparado con un libro o publicación?
- [?] ¿Los detalles no dificultan el razonamiento de alto nivel?
- [?] ¿Formalizar una demostración es "hacer matemática"?

Material para mirar:

- *QED manifesto.*
- *What if Current Foundations of Mathematics are Inconsistent?*
- Publicaciones de Freek Wiedijk y Jeremy Avigad.
- Libro *Homotopy Type Theory*.

Motivaciones

- Demostraciones cada vez más largas y complejas.
- Los humanos somos falibles
  (pregúntenle a cualquier programador).
- Una demostración se acepta cuando nadie encuentra errores.
- ¿Es admisible esta situación?

While the process of finding a proof may require creative insight, the activity of checking a given mathematical argument is an objective activity; mathematical correctness should not be decided by a social process. –T. Coquand

Motivaciones

- Detalles completos: no hay ambigüedades, "mentiras", *hand-waving*, definiciones confusas.
- Las herramientas importan (*cf.* lápiz y papel).
- Queremos aprovechar el poder de cómputo.
- Aliviana la carga de revisar los detalles técnicos.
- Permite certificar que un programa es correcto.
- Puede acercar la matemática a una formulación constructiva. (Extracción de programas).
- Permitiría compilar una base de datos de teoremas.
- Preservación del saber matemático para la posteridad.
- Uso pedagógico como herramienta de estudio.

Formal proof is not merely a method to make absolutely sure we have not made a mistake in a proof, but also a tool that shows us and compels us to understand why a proof works. –G. Gonthier

Coq

- Hay decenas de asistentes de demostración.
- Basados en distintos formalismos, proveen distintas funcionalidades.
- **Coq** es uno de los más populares.
- Basado en teoría de tipos constructiva.
- Varias formalizaciones épicas:
  - Teorema de los cuatro colores (Gonthier, 2005).
  - CompCert (Leroy, 2008).
  - Feit-Thompson (Gonthier, 2012).

¿Qué es un tipo? – El tipo Set

Los tipos son objetos sintácticos definidos inductivamente.

- **Axioma:** Set es un tipo (el tipo de los conjuntos).

Se define una relación binaria "x es un habitante del tipo A", o simplemente "x es de tipo A", notada $x : A$ tal que:

Cada vez que vale...	**...se puede concluir**
$A : \mathsf{Set} \quad B : \mathsf{Set}$	$A \times B : \mathsf{Set}$
$x : A \quad y : B$	$(x, y) : A \times B$
$x : A \times B$	$\mathsf{fst}\ x : A \quad \mathsf{snd}\ x : B$
$A : \mathsf{Set} \quad B : \mathsf{Set}$	$A \to B : \mathsf{Set}$
$f : A \to B \quad x : A$	$f\ x : B$
$y : B$ asumiendo $x : A$	$\mathbf{fun}\ x \Rightarrow y : A \to B$

Notar que los habitantes de Set son, a su vez, tipos.
Intuitivamente:

$$x : A \text{ con } A : \mathsf{Set} \quad \text{sii} \quad "x \in A"$$

Cargar archivo `Basico1.v`.

¿Qué es un tipo? – El tipo Prop

Los tipos también sirven para modelar proposiciones.

- **Axioma:** Prop es un tipo (el tipo de las proposiciones).

Cada vez que vale...	**...se puede concluir**
P : Prop Q : Prop	$P \wedge Q$: Prop
$x : P$ $y : Q$	conj x y : $P \wedge Q$
$x : P \wedge Q$	proj1 x : P proj2 x : Q
P : Prop Q : Prop	$P \to Q$: Prop
$f : P \to Q$ $x : P$	$f\ x : Q$
$y : Q$ asumiendo $x : P$	**fun** $x \Rightarrow y : P \to Q$

Los habitantes de Prop también son tipos.
Intuitivamente:

$x : P$ con P : Prop sii "x es evidencia de que vale P"

"x es una demostración de P"

Ejemplo. Demostrar que vale $P \to (Q \to (P \wedge Q))$.
Cargar archivo `Basico2.v`.

Funciones dependientes (Π)

No todas las funciones que a uno le interesan tienen tipo $A \to B$.
A veces una función f es tal que el tipo de $f\ x$ depende de x.

Un ejemplo:

- La función neutro que devuelve el 1 de un grupo:

$$\text{Grupo} : \text{Set} \quad \text{carrier} : \text{Grupo} \to \text{Set} \quad G : \text{Grupo}$$

$$\text{neutro}\ G : \text{carrier}\ G$$

$$\text{neutro} : \text{Grupo} \to \text{carrier}\ ??$$

- En la práctica humana se comete un abuso de notación escribiendo 1_G (o simplemente 1), pero hay que explicitar las reglas si uno quiere hacer programas que manipulen esta clase de entidades.
- La matemática moderna está impregnada de funciones dependientes.
- (Y de abusos de notación no triviales).

Funciones dependientes (Π)

Para trabajar con funciones dependientes se refinan las reglas para admitir tipos de la forma $\Pi(a : A).\ B_a$ donde B_a puede depender de una variable $a : A$.

Cada vez que vale...	**...se puede concluir**
$A : \text{Type}$ $B_a : \text{Type}$ asumiendo $a : A$	$\Pi(a : A).\ B_a : \text{Type}$
$f : \Pi(a : A).\ B_a \quad x : A$	$f\ x : B_x$
$y : B_x$ asumiendo $x : A$	$\textbf{fun}\ x \Rightarrow y : \Pi(a : A).\ B_a$

Ahora sí se le puede dar tipo a neutro:

$$\text{neutro} : \Pi(G : \text{Grupo}).\ \text{carrier}\ G$$

Notación en Coq.

$$\Pi(a : A).\ B_a = \text{forall}\ a : A,\ B_a$$

Funciones dependientes (Π)

- $f : \Pi(n : \mathbb{N}).\ \mathbb{N} \quad = \quad \mathbb{N} \to \mathbb{N}$
  f toma un $n : \mathbb{N}$ y devuelve un $\mathbb{N}$.
- $f : \Pi(n : \mathbb{N}).\ \mathbb{R}^n$
  f toma un $n : \mathbb{N}$ y devuelve un vector de $\mathbb{R}^n$.
- $f : \Pi(n : \mathbb{N}).\ \text{es\_par}(2 \cdot n)$ suponiendo $\text{es\_par} : \mathbb{N} \to \text{Prop}$.
  f toma un $n : \mathbb{N}$ y devuelve evidencia de que $2 \cdot n$ es par.
  O sea, f es evidencia de que $\forall(n : \mathbb{N}).\ \text{es\_par}(2 \cdot n)$.

Ejemplo. Modelar los números naturales y la noción de paridad.
Cargar archivo `Naturales.v`.

Pares dependientes (Σ)

Dualmente al caso de las funciones, no todos los pares que a uno le interesan tienen tipo $A \times B$.
A veces un par (x, y) es tal que el tipo de y depende de x.

Ejemplo dual al del neutro del grupo:

- Un *pointed set* es un par (X, x_0) con $x_0 \in X$.
- ¿Cómo expresamos esto usando tipos?

$$X : \mathsf{Set} \quad x_0 : X$$

$$(X, x_0) : \mathsf{Set} \times X$$

- ¿Cómo definimos el tipo PointedSet?

$$\mathsf{PointedSet} := \mathsf{Set} \times \; ??$$

Pares dependientes (Σ)

Para trabajar con pares dependientes se refinan las reglas para admitir tipos de la forma $\Sigma(a : A).\ B_a$ donde B_a puede depender de una variable $a : A$.

Cada vez que vale...	**...se puede concluir**
$A : \mathsf{Type}$ $B_a : \mathsf{Type}$ asumiendo $a : A$	$\Sigma(a : A).\ B_a \ : \ \mathsf{Type}$
$x : A \quad y : B_x$	$\mathsf{exist}_B\ x\ y : \Sigma(a : A).\ B_a$
$x : \Sigma(a : A).\ B_a$	$\pi_1\ x : A \quad \pi_2\ x : B_{(\pi_1\ x)}$

Ahora sí se puede definir PointedSet:

$$\mathsf{PointedSet} := \Sigma(X : \mathsf{Set}).\ X$$

Notación en Coq. Varios sabores (y muchos dolores de cabeza):

$\Sigma(a : A).\ B_a = \mathsf{exists}\ a : A,\ B_a$ no constructivo (es una Prop)
$\Sigma(a : A).\ B_a = \{a : A \mid B_a\}$ es un Set cuando $B_a : \mathsf{Prop}$
$\Sigma(a : A).\ B_a = \{a : A\ \&\ B_a\}$ es un Set cuando $B_a : \mathsf{Set}$

Pares dependientes (Σ)

- $p : \Sigma(n : \mathbb{N}).\ \mathbb{N} \quad = \quad \mathbb{N} \times \mathbb{N}$
  p es un par ordenado de naturales.
- $p : \Sigma(n : \mathbb{N}).\ \mathbb{R}^n$
  p es un par cuya primera componente es un $n : \mathbb{N}$, y la segunda es un vector de $\mathbb{R}^n$.
  Representa un vector de $\mathbb{R}^n$ con el permiso de variar n.
- $p : \Sigma(n : \mathbb{N}).\ \text{es\_primo}(n)$ suponiendo $\text{es\_primo} : \mathbb{N} \to \text{Prop}$.
  p es un par cuya primera componente es un $n : \mathbb{N}$, y la segunda es evidencia de que n es primo.
  O sea, p es evidencia de que $\exists(n : \mathbb{N}).\ \text{es\_primo}(n)$.

Ejemplo. Modelar el conjunto de naturales mayores que 10.
Cargar archivo `Sigma.v`.

Más ejemplos:

- Propiedades de relaciones: archivo `Relaciones.v`
- Inducción estructural: archivo `Induccion.v`
- Teorema de Cayley: archivo `Cayley.v`

Referencias

Material filosófico / motivaciones:

[1] *QED manifesto*
`http://www.cs.ru.nl/~freek/qed/qed.html`

[2] Vladimir Voevodsky.
What if Current Foundations of Mathematics are Inconsistent?
`https://video.ias.edu/voevodsky-80th`

[3] Notas de Freek Wiedijk.
`http://www.cs.ru.nl/~freek/notes/index.html`

[4] Publicaciones de Jeremy Avigad.
`http://www.andrew.cmu.edu/user/avigad/`

Referencias

Material de referencia:

[5] Yves Bertot, Pierre Casteran.
Interactive Theorem Proving and Program Development

[6] Benjamin Pierce *et al.*
Software Foundations

[7] Adam Chlipala.
Certified Programming with Dependent Types

[8] The Univalent Foundations Program.
Homotopy Type Theory: Univalent Foundations of Mathematics
`http://homotopytypetheory.org/book/`