PPA

Un asistente de demostración para lógica de primer orden con extracción de testigos usando la traducción de Friedman

Manuel Panichelli

Deparatamento de Computación, FCEyN, UBA

Diciembre 2024

Introducción

Asistentes de demostración

- Los **asistentes de demostración** son herramientas que facilitan la escritura y el chequeo de demostraciones por computadora.
- Usos usuales: formalización de teoremas matemáticos y verificación de programas.
- Ventajas:[1]
  - Facilitan la colaboración a gran escala (mediante la confianza en el asistente).
  - Habilitan generación automática de demostraciones con IA. Por ej. un *LLM* (como *ChatGPT*) suele devolver alucinaciones, que pueden ser filtradas automáticamente con un asistente.

[1] Terrence Tao - Machine Assisted Proof

Asistentes de demostración

Implementan distintas *teorías*. (TODO: No me gusta teoria, se usa para teorias de primer orden) Ejemplos:

- Mizar (lógica de primer orden)
- Coq (teoría de tipos)
- Agda (teoría de tipos)
- Isabelle (lógica de orden superior / teoría de conjuntos ZF)

PPA

- Diseñamos e implementamos **PPA** (*Pani's Proof Assistant*), un asistente de demostración para lógica **clásica** de primer orden.
- Permite hacer extracción de testigos: dada una demostración de $\exists x.p(x)$, encuentra t tal que $p(t)$.
- Aporte principal: implementación de extracción de testigos para lógica clásica de forma directa (vemos los detalles después).

Representación de demostraciones

Queremos escribir demostraciones en la computadora. ¿Cómo las representamos?. Veamos un ejemplo.

- Tenemos dos premisas
  1. Los alumnos que faltan a los exámenes, los reprueban.
  2. Si se reprueba un final, se recursa la materia.
- A partir de ellas, podríamos demostrar que si un alumno falta a un final, entonces recursa la materia.

Teorema

Si ((falta entonces reprueba) y (reprueba entonces recursa)) y falta, entonces recursa

Demostración

- Asumo que falta. Quiero ver que recursa.
- Sabemos que si falta, entonces reprueba. Por lo tanto reprobó.
- Sabemos que si reprueba, entonces recursa. Por lo tanto recursó. □

Sistemas deductivos

- La demostración anterior es poco precisa. No se puede representar rigurosamente.
- Necesitamos **sistemas deductivos**: sistemas lógicos formales usados para demostrar setencias. Pueden ser representados como un tipo abstracto de datos.
- Usamos **deducción natural**. Compuesto por,
  - **Lenguaje formal**: lógica de primer orden.
  - **Reglas de inferencia**: lista de reglas que se usan para probar teoremas a partir de axiomas y otros teoremas. Por ejemplo, *modus ponens* (si es cierto $A \to B$ y A, se puede concluir B) o *modus tollens* (si es cierto $A \to B$ y $\neg B$, se puede concluir $\neg A$)
  - **Axiomas**: fórmulas de L que se asumen válidas. Todos los teoremas se derivan de axiomas. Se usan para modelar *teorías* de primer orden (por ej. teoría de estudiantes en la facultad).

Lógica de primer orden

Definición (Términos)

Los términos están dados por la gramática:

$$\begin{aligned} t ::= {} & x && \text{(variables)} \\ & \mid f(t_1, \ldots, t_n) && \text{(funciones)} \end{aligned}$$

Definición (Fórmulas)

Las fórmulas están dadas por la gramática:

$$\begin{aligned} A, B ::= {} & p(t_1, \ldots, t_n) && \text{(predicados)} \\ & \mid \bot \mid \top && \text{(falso o \textit{bottom} y verdadero o \textit{top})} \\ & \mid A \wedge B \mid A \vee B && \text{(conjunción y disyunción)} \\ & \mid A \to B \mid \neg A && \text{(implicación y negación)} \\ & \mid \forall x.A \mid \exists x.A && \text{(cuantificador universal y existencial)} \end{aligned}$$

Los predicados son **fórmulas atómicas**. Los de aridad 0 además son llamados *variables proposicionales*.

Notación

Usamos

- $x, y, z, \ldots$ como **variables**.
- $f, g, h, \ldots$ como **símbolos de función**.
- $p, q, r, \ldots$ como **símbolos de predicado**.
- $t, u, \ldots$ para referirnos a **términos**.
- $a, b, c, \ldots, A, B, C, \ldots$ y $\varphi, \psi, \ldots$ para referirnos a **fórmulas**.

Deducción natural

Deducción natural

Definiciones

- Γ es un **contexto de demostración**, conjunto de fórmulas que se asumen válidas
- Notación: $\Gamma, \varphi = \Gamma \cup \{\varphi\}$
- $\vdash$ es la **relación de derivabilidad** definida a partir de las *reglas de inferencia*. Permite escribir juicios $\Gamma \vdash \varphi$.
- Intuición: *"φ es una consecuencia de las suposiciones de Γ"*
- El juicio es cierto si en una cantidad finita de pasos podemos concluir φ a partir de las fórmulas de Γ, los axiomas y las reglas de inferencia.
- Decimos que φ es *derivable* a partir de Γ.

Reglas de inferencia

Definición (Reglas de inferencia)

$$\frac{}{\Gamma, A \vdash A}\ \mathrm{Ax} \qquad \frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \wedge B}\ \mathrm{I}\wedge$$

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \to B}\ \mathrm{I}\!\to \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A}\ \mathrm{E}\wedge_1$$

$$\frac{\Gamma \vdash A \to B \qquad \Gamma \vdash A}{\Gamma \vdash B}\ \mathrm{E}\!\to \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash B}\ \mathrm{E}\wedge_2$$

Dos tipos para cada conectivo y cuantificador, dada una fórmula formada con un conectivo:

- **Introducción**: ¿Cómo la demuestro?
- **Eliminación**: ¿Cómo la uso para demostrar otra?

Ejemplo

Vamos a demostrar el ejemplo informal en deducción natural. Lo modelamos para un alumno y materia particulares. Notamos:

- $X \equiv \mathrm{reprueba}(\mathit{juan}, \mathrm{final}(\mathit{logica}))$
- $R \equiv \mathrm{recursa}(\mathit{juan}, \mathit{logica})$
- $F \equiv \mathrm{falta}(\mathit{juan}, \mathrm{final}(\mathit{logica}))$

Queremos probar entonces

$$\Big((F \to X) \land (X \to R)\Big) \to (F \to R)$$

Ejemplo

$$
\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \vdash (F \to X) \wedge (X \to R)}\ \mathsf{Ax}}{\Gamma \vdash X \to R}\ \mathsf{E}\wedge_1 \qquad \begin{array}{c}\Pi\\ \Gamma \vdash X\end{array}}{\Gamma = (F \to X) \wedge (X \to R),\ F \vdash R}\ \mathsf{E}{\to}}{\dfrac{(F \to X) \wedge (X \to R) \vdash F \to R}{\vdash \Big((F \to X) \wedge (X \to R)\Big) \to (F \to R)}\ \mathsf{I}{\to}}\ \mathsf{I}{\to}
$$

donde

$$
\Pi = \dfrac{\dfrac{\dfrac{}{\Gamma \vdash (F \to X) \wedge (X \to R)}\ \mathsf{Ax}}{\Gamma \vdash F \to X}\ \mathsf{E}\wedge_2 \qquad \dfrac{}{\Gamma \vdash F}\ \mathsf{Ax}}{\Gamma \vdash X}\ \mathsf{E}{\to}
$$

Reglas de inferencia

Definición (Reglas de inferencia)

$$\frac{}{\Gamma \vdash A \vee \neg A}\ \text{LEM}$$

$$\frac{\Gamma \vdash \bot}{\Gamma \vdash A}\ \text{E}\bot \qquad \frac{\Gamma, A \vdash \bot}{\Gamma \vdash \neg A}\ \text{I}\neg$$

$$\frac{}{\Gamma \vdash \top}\ \text{I}\top \qquad \frac{\Gamma \vdash \neg A \qquad \Gamma \vdash A}{\Gamma \vdash \bot}\ \text{E}\neg$$

$$\frac{\Gamma \vdash A}{\Gamma \vdash A \vee B}\ \text{I}\vee_1 \qquad \frac{\Gamma \vdash B}{\Gamma \vdash A \vee B}\ \text{I}\vee_2$$

$$\frac{\Gamma \vdash A \vee B \qquad \Gamma, A \vdash C \qquad \Gamma, B \vdash C}{\Gamma \vdash C}\ \text{E}\vee$$

Reglas de inferencia

Definición (Sustitución)

Notamos como $A\{x := t\}$ a la sustitución de todas las ocurrencias libres de la variable x por el término t en la fórmula A.

Definición (Reglas de cuantificadores)

$$\frac{\Gamma \vdash A \qquad x \notin fv(\Gamma)}{\Gamma \vdash \forall x.A}\ \mathsf{I}\forall \qquad\qquad \frac{\Gamma \vdash \forall x.A}{\Gamma \vdash A\{x := t\}}\ \mathsf{E}\forall$$

$$\frac{\Gamma \vdash A\{x := t\}}{\Gamma \vdash \exists x.A}\ \mathsf{I}\exists$$

$$\frac{\Gamma \vdash \exists x.A \qquad \Gamma, A \vdash B \qquad x \notin fv(\Gamma, B)}{\Gamma \vdash B}\ \mathsf{E}\exists$$

Reglas admisibles

- Mencionamos *modus tollens* pero no aparece en las reglas de inferencia.
- Queremos un sistema lógico **minimal**: no agregamos como regla de inferencia lo que podemos derivar a partir de las existentes, las reglas **admisibles**.
- Se implementan como *macros*: cada uso de la regla admisible se reemplaza por su demostración.

Lema (Modus tollens)

$$
\dfrac{
  \dfrac{
    \dfrac{
      \dfrac{}{\Gamma \vdash (A \to B) \wedge \neg B}\,Ax
    }{\Gamma \vdash \neg B}\,E\wedge_2
    \qquad
    \dfrac{
      \dfrac{
        \dfrac{}{\Gamma \vdash (A \to B) \wedge \neg B}\,Ax
      }{\Gamma \vdash A \to B}\,E\wedge_1
      \quad
      \dfrac{}{\Gamma \vdash A}\,Ax
    }{\Gamma \vdash B}\,E\!\to
  }{\Gamma = (A \to B) \wedge \neg B, A \vdash \bot}\,E\neg
}{
  \dfrac{(A \to B) \wedge \neg B \vdash \neg A}{\vdash (A \to B \wedge \neg B) \to \neg A}\,I\!\to
}
$$

Sustitución sin capturas

Para la sustitución $A\{x := t\}$ queremos evitar la **captura de variables**, por ejemplo

$$(\forall y.p(x))\{x := y\} \stackrel{?}{=} \forall y.p(y)$$

sustituyendo sin más, capturamos a la variable x que ahora está ligada. Lo evitamos **automáticamente**: cuando se encuentra con una captura, se renombra la variable ligada de forma que no ocurra

$$(\forall y.p(x))\{x := y\} = \forall z.p(y)$$

Alfa equivalencia

- Si tenemos una hipótesis $\exists x.p(x)$ queremos poder usarla para demostrar $\exists y.p(y)$.
- No son iguales, pero son α-**equivalentes**: si renombramos variables ligadas de forma apropiada, son iguales.
- Algoritmo naíf: cuadrático en la estructura de la fórmula, renombrando recursivamente.
- Algoritmo cuasilineal: manteniendo dos sustituciones, una por fórmula.

Ejemplo

$$
\begin{array}{rlr}
(\exists x.f(x)) \overset{\alpha}{=} (\exists y.f(y)) & & \{\},\{\} \\
\iff & f(x) \overset{\alpha}{=} f(y) & \{x \mapsto z\},\{y \mapsto z\} \\
\iff & x \overset{\alpha}{=} y & \{x \mapsto z\},\{y \mapsto z\} \\
\iff & z = z. &
\end{array}
$$

PPA

Mathematical vernacular

Aparentemente hay una forma canónica de presentar demostraciones matemáticas[2]. Descubierta e implementada independientemente en Mizar, Isar (Isabelle), etc. Combinación de ideas:

- **Deducción natural en estilo de *Fitch***. Notación equivalente en la cual las demostraciones son representadas como listas de fórmulas en lugar de árboles. Las que aparecen antes justifican las que aparecen después.
- **Reglas de inferencia *"big step"***: una forma de afirmar que $A_1, \ldots, A_n \vdash A$ es válida, sin tener que demostrarlo a mano.
- **Sintaxis similar a un lenguaje de programación** en lugar del lenguaje natural usado para demostraciones.

[2] *Mathematical Vernacular* de Freek Wiedijk

PPA

Diseñamos e implementamos el *lenguaje* PPA, inspirado en el *mathematical vernacular*. Veamos la **interfaz** completa y luego la implementación.

Ejemplo demostración

```
axiom falta_reprueba: forall A . forall E .
    falta(A, E) -> reprueba(A, E)
axiom reprueba_recursa: forall A . forall M .
    reprueba(A, final(M)) -> recursa(A, M)

theorem falta_entonces_recursa: forall A . forall M .
    falta(A, final(M)) -> recursa(A, M)
proof
    let A
    let M
    suppose falta: falta(A, final(M))
    have reprueba: reprueba(A, final(M)) by falta, falta_reprueba
    thus recursa(A, M) by reprueba, reprueba_recursa
end
```

Programas

Un **programa** de PPA consiste en una lista de **declaraciones**, que pueden ser

- **Axiomas**: fórmulas que se asumen válidas

```
axiom <name> : <form>
```

- **Teoremas**: fórmulas junto con sus demostraciones.

```
theorem <name> : <form>
proof
   <steps>
end
```

Identificadores

- **Variables** (<var>)

```
(\_|[A-Z])[a-zA-Z0-9\_\-]*(\')*
```

- **Identificadores** (<id>)

```
[a-zA-Z0-9\_\-\?!#\$\%\*\+\<\>\=\?\@\^]+(\')*
```

- **Nombres** (<name>)
  Pueden ser identificadores o strings arbitrarios encerrados por comillas dobles.

```
<id> | \"[^\"]*\"
```

Fórmulas y términos

Términos:

- Variables: `<var>`
- Funciones: `<id>(<term>, ..., <term>)`

Funciones:

- Predicados: `<id>(<term>, ..., <term>)`
- `<form> & <form>`
- `<form> | <form>`
- `<form> -> <form>`
- `<form> <-> <form>`
- `~ <form>`
- **`exists`** `<var> . <form>`
- **`forall`** `<var> . <form>`
- **`true`**, **`false`**
- `(<form>)`

Demostraciones

- Lista de comandos que reducen sucesivamente la *tesis* (fórmula a demostrar) hasta agotarla por completo.
- Corresponden aproximadamente a reglas de inferencia de deducción natural (vistas como una demostración en el estilo de Fitch).
- Tienen disponible un **contexto** con todas las hipótesis asumidas (como axiomas) o demostradas (teoremas y comandos que demuestran hipótesis auxiliares).

by - El mecanismo principal de demostración

- `<form>` **`by`** `<h1>, ..., <hn>` afirma que la fórmula es una consecuencia lógica de las fórmulas que corresponden a las hipótesis provistas.
- Los nombres de las hipótesis son del tipo `<name>` (o bien identificadores o *strings* arbitrarios).
- Por debajo usa un *solver* completo para lógica proposicional pero heurístico para primer órden.
- Se usa para eliminar implicaciones y universales.
- Usado por dos comandos principales: **`thus`** y **`have`**.

Thus

```
thus <form> by <h1>, ..., <hn>
```

Si `<form>` es *parte* de la tesis, y el *solver* puede demostrar la implicación, lo demuestra automáticamente y lo descarga de la tesis.

Eliminación de implicación

```
axiom ax1: a -> b
axiom ax2: b -> c

theorem t1: a -> c
proof
   suppose a: a

   // La tesis ahora es c
   thus c by a, ax1, ax2
end
```

Eliminación de universal

```
axiom ax: forall X . f(X)

theorem t: f(n)
proof
   thus f(n) by ax
end
```

Have

```
have <name>: <form> by <h1>, ..., <hn>
```

Análogo a **thus**, pero introduce una afirmación *auxiliar* sin reducir la tesis, agregándola al contexto.

Eliminación de implicación en dos pasos

```
axiom ax1: a -> b
axiom ax2: b -> c

theorem t1: a -> c
proof
    suppose a: a
    have b: b by a, ax1
    thus c by b, ax2
end
```

Hipótesis anterior

Ambas pueden referirse a la hipótesis anterior con guión medio (`-`), y pueden hacerlo implícitamente usando **`hence`** y **`have`**.

Comando	Alternativo	¿Reduce la tesis?
`thus`	**`hence`**	Sí
`have`	**`then`**	No

Eliminación en dos pasos

```
axiom ax1: a -> b
axiom ax2: b -> c

theorem t1: a -> c
proof
   suppose a: a
   have b: b by a, ax1
   thus c by b, ax2
end
```

Alternativas equivalentes

```
proof
   suppose a: a
   have b: b by -, ax1
   thus c by -, ax2
end
proof
   suppose -: a
   then -: b by ax1
   hence c by ax2
end
```

By opcional

- El **by** es opcional
- Si se omite, la fórmula debe ser demostrable por el *solver* sin partir de ninguna hipótesis
- Vale para todas las tautologías proposicionales.

Tautología proposicional

```
theorem "distributiva de negación sobre disyunción":
   ~(a | b) <-> ~a & ~b
proof
   thus ~(a | b) <-> ~a & ~b
end
```

Comandos y reglas de inferencia

Regla	Comando
LEM	**`cases`**
Ax	**`by`**
$\mathsf{I}\exists$	**`take`**
$\mathsf{E}\exists$	**`consider`**
$\mathsf{I}\forall$	**`let`**
$\mathsf{E}\forall$	**`by`**
$\mathsf{I}\vee_1$	**`by`**
$\mathsf{I}\vee_2$	**`by`**
$\mathsf{E}\vee$	**`cases`**

Regla	Comando
$\mathsf{I}\wedge$	**`by`**
$\mathsf{E}\wedge_1$	**`by`**
$\mathsf{E}\wedge_2$	**`by`**
$\mathsf{I}\rightarrow$	**`suppose`**
$\mathsf{E}\rightarrow$	**`by`**
$\mathsf{I}\neg$	**`suppose`**
$\mathsf{E}\neg$	**`by`**
$\mathsf{I}\top$	**`by`**
$\mathsf{E}\bot$	**`by`**

Suppose (I→/ I¬)

suppose <name>: <form> (I→/ I¬)

- Si la tesis es una implicación $A \to B$, agrega el antecedente A como hipótesis con el nombre dado y reduce la tesis al consecuente B
- Viendo la negación como una implicación $\neg A \equiv A \to \bot$, permite introducir negaciones, tomando $B = \bot$.

Introducción de implicación

```
theorem "suppose":
    a -> (a -> b) -> b
proof
    suppose h1: a
    suppose h2: a -> b
    thus b by h1, h2
end
```

Introducción de negación

```
theorem "not intro":
    ~b & (a -> b) -> ~a
proof
    suppose h: ~b & (a -> b)
    suppose a: a
    hence false by h, a
end
```

Cases (E$\vee$)

cases by <h1>, ..., <hn> (I$\rightarrow$/ I$\neg$)

- Permite razonar por casos a partir de una disyunción. Para cada uno, se debe demostrar la tesis en su totalidad.
- Si los casos son <f1> a <fn>, tiene que valer <f1> | ... | <fn> **by** <h1>, ..., <hn>.
- Se puede omitir el **by** para razonar mediante LEM (casos φ y $\neg\varphi$).

Cases

```
theorem "cases":
    (a & b) | (c & a) -> a
proof
    suppose h: (a & b) | (c & a)
    cases by h
        case a & b
            hence a
        case right: a & c
            thus a by right
    end
end
```

Take (I∃)

take `<var> := <term>` (I∃)

- Introduce un existencial instanciando su variable y reemplazándola por un término.
- Si la tesis es **exists** `X . p(X)`, luego de **take** `X := a`, se reduce a `p(a)`.

Consider (E∃)

consider `<var>` **st** `<name>: <form>` **by** `<h1>, ..., <hn>` (E∃)

- Si se puede justificar **exists** `X. p(X)`, permite razonar sobre tal `X`.
- Agrega `<form>` como hipótesis al contexto, con nombre `<name>`. No reduce la tesis.
- Debe valer **exists** `<var> . <form>` **by** `<h1>, ..., <hn>`
- Permite α-equivalencias: Si podemos justificar **exists** `X. p(X)`, podemos usarlo como **consider** `Y` **st** `h: p(Y)` **by** `...`.

Let (I∀)

`let` `<var>` (I∀)

- Permite demostrar un cuantificador universal.
- Si la tesis es **`forall`** `X . p(X)`, luego de **`let`** `X`, la tesis se reduce a `p(X)`.
- Permite renombrar la variable, por ejemplo luego de **`let`** `Y` la tesis se reduce a `p(Y)`.

Descarga de conjunciones

Si la tesis es una conjunción, se puede probar un subconjunto de ella y se reduce el resto.

Descarga simple

```
theorem "and discharge":
    a -> b -> (a & b)
proof
    suppose "a" : a
    suppose "b" : b
    // La tesis es a & b
    hence b by "b"

    // La tesis es a
    thus a by "a"
end
```

Descarga compleja

```
axiom "a": a
axiom "b": b
axiom "c": c
axiom "d": d
axiom "e": e
theorem "and discharge":
    (a & b) & ((c & d) & e)
proof
    thus a & e by "a", "e"
    thus d by "d"
    thus b & c by "b", "c"
end
```

Equivalently

```
equivalently <form>
```

- Permite reducir la tesis a una fórmula equivalente
- Se puede usar por ejemplo para descarga de conjunciones, o para razonar por el absurdo mediante la eliminación de la doble negación.

Descarga de conjunción

```
axiom a1: ~a
axiom a2: ~b

theorem "ejemplo" : ~(a | b)
proof
   equivalently ~a & ~b
   thus ~a by a1
   thus ~b by a2
end
```

Razonamiento por el absurdo

```
theorem t: <form>
proof
   equivalently ~~<form>
   suppose <name>: ~<form>
   // Demostración de <form>
   // por el absurdo,
   // asumiendo ~<form>
   // y llegando a una
   // contradicción (false).
end
```

Claim

```
claim <name>: <form>
```

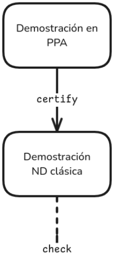
Permite demostrar una afirmación auxiliar. Útil para ordenar las demostraciones sin tener que definir otro teorema.

```
theorem t: <form1>
proof
   claim <name>: <form2>
   proof
      // Demostración de <form2>.
   end
   // Demostración de <form1> refiriéndose a <name>.
end
```

Certificador

Certificados

- Las demostraciones de PPAse *certifican* generando una demostración de deducción natural.
- Cumple con el criterio de de bruijn



Extracción de testigos

Introducción

- PPA (*Pani's proof assistant*) es un **asistente de demostraciones** inspirado en Mizar.
- Es un **lenguaje de programación** implementado en Haskell que permite escribir y chequear demostraciones en lógica *clásica* de primer órden.
- A diferencia de Prolog, **no demuestra todo automáticamente***. Deben ser escritas *rigurosamente* por el usuario.
- (WIP) permite la extracción de testigos mediante la **traducción de Friedman**.

Asistentes de demostraciones (*proof assistants*)

- Son programas que *asisten* al usuario a la hora de escribir demostraciones, permiten representarlas en un programa
- Aplicaciones: Formalización de teoremas, verificación formal de programas, etc.
- Ejemplos: Coq, Isabelle (Isar), **Mizar**, ...
- Ventajas[3]:
  - facilitan colaboración a gran escala (via confianza en el checker)
  - habilitan generación automática de demostraciones con ML. Un LLM suele devolver alucinaciones, pero pueden ser chequeadas

[3] Terrence Tao - Machine Assisted Proof

Arquitectura de PPA

¿Por qué certificados?

- Si formalizamos una demostración en PPA y queremos chequear que sea correcta, hay que confiar en la implementación del *proof assistant*.
- **Criterio de De Brujin**: si guardamos una demostración de bajo nivel de forma completa, puede ser chequeada por un programa independiente (que es sencillo de implementar).
- Cumplida por Coq, pero no Mizar[4].

[4] Adam Naumowicz - A brief overview of Mizar

PPA

- Es un lenguaje. Frontend implementado con un *parser generator* (happy + alex)
- Permite definir axiomas y teoremas con sus demostraciones, que al **certificarse** generan una demostración en deducción natural.
- Basado en *Mathematical Vernacular*[5]: un leguaje formal para escribir demostraciones similar al natural.

[5] The Mathematical Vernacular - Freek Wiedijk

Ejemplo

Una demostración es una secuencia de *comandos*, que pueden ir sucesivamente reduciendo la *tesis* (objetivo a probar) y agregando hipótesis a un contexto. Se mapean a reglas de deducción natural.

Teorema

```
theorem "implication transitivity":
    (a -> b) & (b -> c) -> (a -> c) // Tesis
proof
    suppose h1: (a -> b) & (b -> c)
    // Tesis: a -> c

    suppose h2: a
    // Tesis: c

    thus c by h1, h2
end
```

by

- El mecanismo principal para demostrar es el **by**, que *automáticamente* demuestra que un hecho es consecuencia de una lista de hipótesis.
- Se usa en lugar de E$\rightarrow$ y E$\forall$
- Es completo para lógica proposicional pero heurístico para LPO

Ejemplo

```
axoim ax1: a -> b
axiom ax2: a
theorem thm: b
proof
    thus b by ax1, ax2
end
```

Para demostrar $\big((a \to b) \land a\big) \to b$ lo hacemos por el absurdo: negamos y encontramos una contradicción.

DNF

Primero convertimos la fórmula a forma normal disyuntiva (DNF)

$$
\begin{array}{ll}
\neg\big[\big((a \to b) \wedge a\big) \to b\big] & \\
\equiv \neg\big[\neg\big((a \to b) \wedge a\big) \vee b\big] & (x \to y \equiv \neg x \vee y) \\
\equiv \neg\neg\big((a \to b) \wedge a\big) \wedge \neg b & (\neg(x \vee y) \equiv \neg x \wedge \neg y) \\
\equiv \big((a \to b) \wedge a\big) \wedge \neg b & (\neg\neg x \equiv x) \\
\equiv (\neg a \vee b) \wedge a \wedge \neg b & (x \to y \equiv \neg x \vee y) \\
\equiv (\neg a \vee b) \wedge a \wedge \neg b & ((x \vee y) \wedge z \equiv (x \wedge z) \vee (y \wedge z)) \\
\equiv (\neg a \wedge a \wedge \neg b) \vee (b \wedge a \wedge \neg b) &
\end{array}
$$

Contradicción

Ya tenemos la fórmula en DNF, ahora tenemos que demostrar la contradicción. Lo hacemos refutando cada cláusula

$$(\neg a \land a \land \neg b) \lor (b \land a \land \neg b) \vdash \bot$$

Reglas

$$\frac{\Gamma \vdash A \lor B \qquad \Gamma, A \vdash C \qquad \Gamma, B \vdash C}{\Gamma \vdash C}\ \mathrm{E}\lor$$

$$\frac{\Gamma \vdash \neg A \qquad \Gamma \vdash A}{\Gamma \vdash \bot}\ \mathrm{E}\neg$$

Recap by

Teniendo en el contexto $\Gamma = \{h_1 : b_1, \ldots, h_n : b_n\}$ para certificar

$$\text{thus } a \text{ by } h_1 \ldots h_n$$

- Debe demostrar $b_1 \land \ldots \land b_n \to a$
- Lo hace por absurdo: la niega y encuentra una contradicción
- Primero la convierte a forma normal disyuntiva (DNF)
- Luego refuta cada cláusula (conjunción de literales)
  - False $(\bot \land p \land q)$
  - Literales opuestos $(p(a) \land \neg p(a) \land q)$
  - Eliminación de existencial $(\forall x.p(x) \land \neg p(a))$

Desafío: ¡Hay que generar una demostración de deducción natural!

DNF

¿Cómo demostramos el pasaje de uno al otro?

$$\neg[((a \to b) \land a) \to b] \vdash \bot$$

$$\vdots$$

$$(\neg a \land a \land \neg b) \lor (b \land a \land \neg b) \vdash \bot$$

Generando demostraciones para todas las equivalencias, y convirtiendo la fórmula paso por paso (*"small step"*)

$$\begin{aligned}
\neg\neg x &\equiv x \\
\neg\bot &\equiv \top \\
\neg\top &\equiv \bot \\
x \to y &\equiv \neg x \lor y \\
\neg(x \lor y) &\equiv \neg x \land \neg y \\
\neg(x \land y) &\equiv \neg x \lor \neg y
\end{aligned}$$

$$\begin{aligned}
(x \lor y) \land z &\equiv (x \land z) \lor (y \land z) \\
z \land (x \lor y) &\equiv (z \land x) \lor (z \land y) \\
x \lor (y \lor z) &\equiv (x \lor y) \lor z \\
x \land (y \land z) &\equiv (x \land y) \land z
\end{aligned}$$

DNF

Para poder hacerlo paso por paso también hace falta demostrar la *congruencia* de los operadores

$$a \vee \neg(b \vee c) \equiv a \vee (\neg b \wedge \neg c)$$

En general,

$$\alpha \equiv \alpha' \Rightarrow \alpha \wedge \beta \equiv \alpha' \wedge \beta$$
$$\beta \equiv \beta' \Rightarrow \alpha \wedge \beta \equiv \alpha \wedge \beta'$$

Análogo para $\vee$, $\neg$

¿Por qué este mecanismo?

- Es un procedimiento completo para LP pero **heurístico** para LPO, puede fallar (i.e no demuestra cualquier cosa)
- Satisfacibilidad de LPO es indecidible
- Mecanismos como *resolución general* se pueden colgar
- Podríamos haber hecho otro, queríamos hacer *alguno*

Friedman

Lógica clásica / Lógica intuicionista

- La lógica **clásica** no siempre es constructiva, por el *principio del tercero excluido* (LEM):

  para toda proposición A, es verdadera ella o su negación

  $$A \vee \neg A$$

- La lógica **intuicionista** se puede describir de forma sucinta como la lógica clásica sin LEM. Equivalentemente, tampoco vale la *eliminación de la doble negación* ($\neg\neg A \to A$)

Demostración no constructiva

Teorema

Existen dos números irracionales a, b tq a^b es racional.

Sabemos que $\sqrt{2}$ es irracional, y por LEM que $\sqrt{2}^{\sqrt{2}}$ es o racional o irracional.

- Si $\sqrt{2}^{\sqrt{2}}$ es racional, tomamos $a = b = \sqrt{2}$
- Sino, tomamos $a = \sqrt{2}^{\sqrt{2}}$, $b = \sqrt{2}$ y luego

  $$a^b = (\sqrt{2}^{\sqrt{2}})^{\sqrt{2}} = \sqrt{2}^{\sqrt{2}\sqrt{2}} = \sqrt{2}^2 = 2$$

  que es racional.

¡No nos dice cuales son a y b!

Traducción de Friedman

- Queremos “reducir” o “ejecutar” los programas para obtener testigos de existenciales.
- La lógica clásica no es ejecutable (no constructiva). La intuicionista sí
- Friedman traduce de clásica a intuicionista. Caveat: solo fórmulas $\in \Pi^0_2$ (i.e de la forma $\forall x_1 \dots \forall x_n \exists y . \varphi$)

¿En dónde estamos parados?