
Universidad de Buenos Aires
Facultad de Ciencias Exactas y Naturales
Departamento de Computación

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

Tesis de Licenciatura en Ciencias de la Computación

Manuel Panichelli

Director: Pablo Barenbaum

Buenos Aires, 2024

# PPA - UN ASISTENTE DE DEMOSTRACIÓN PARA LÓGICA DE PRIMER ORDEN CON EXTRACCIÓN DE TESTIGOS USANDO LA TRADUCCIÓN DE FRIEDMAN

La princesa Leia, líder del movimiento rebelde que desea reinstaurar la República en la galaxia en los tiempos ominosos del Imperio, es capturada por las malévolas Fuerzas Imperiales, capitaneadas por el implacable Darth Vader. El intrépido Luke Skywalker, ayudado por Han Solo, capitán de la nave espacial "El Halcón Milenario", y los androides, R2D2 y C3PO, serán los encargados de luchar contra el enemigo y rescatar a la princesa para volver a instaurar la justicia en el seno de la Galaxia (aprox. 200 palabras).

**Palabras claves:** Guerra, Rebelión, Wookie, Jedi, Fuerza, Imperio (no menos de 5).

# PPA - A PROOF-ASSISTANT FOR FIRST-ORDER LOGIC WITH WITNESS EXTRACTION USING FRIEDMAN'S TRANSLATION

In a galaxy far, far away, a psychopathic emperor and his most trusted servant – a former Jedi Knight known as Darth Vader – are ruling a universe with fear. They have built a horrifying weapon known as the Death Star, a giant battle station capable of annihilating a world in less than a second. When the Death Star's master plans are captured by the fledgling Rebel Alliance, Vader starts a pursuit of the ship carrying them. A young dissident Senator, Leia Organa, is aboard the ship & puts the plans into a maintenance robot named R2-D2. Although she is captured, the Death Star plans cannot be found, as R2 & his companion, a tall robot named C-3PO, have escaped to the desert world of Tatooine below. Through a series of mishaps, the robots end up in the hands of a farm boy named Luke Skywalker, who lives with his Uncle Owen & Aunt Beru. Owen & Beru are viciously murdered by the Empire's stormtroopers who are trying to recover the plans, and Luke & the robots meet with former Jedi Knight Obi-Wan Kenobi to try to return the plans to Leia Organa's home, Alderaan. After contracting a pilot named Han Solo & his Wookiee companion Chewbacca, they escape an Imperial blockade. But when they reach Alderaan's coordinates, they find it destroyed - by the Death Star. They soon find themselves caught in a tractor beam & pulled into the Death Star. Although they rescue Leia Organa from the Death Star after a series of narrow escapes, Kenobi becomes one with the Force after being killed by his former pupil - Darth Vader. They reach the Alliance's base on Yavin's fourth moon, but the Imperials are in hot pursuit with the Death Star, and plan to annihilate the Rebel base. The Rebels must quickly find a way to eliminate the Death Star before it destroys them as it did Alderaan (aprox. 200 palabras).

**Keywords:** War, Rebellion, Wookie, Jedi, The Force, Empire (no menos de 5).

# AGRADECIMIENTOS

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Fusce sapien ipsum, aliquet eget convallis at, adipiscing non odio. Donec porttitor tincidunt cursus. In tellus dui, varius sed scelerisque faucibus, sagittis non magna. Vestibulum ante ipsum primis in faucibus orci luctus et ultrices posuere cubilia Curae; Mauris et luctus justo. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos himenaeos. Mauris sit amet purus massa, sed sodales justo. Mauris id mi sed orci porttitor dictum. Donec vitae mi non leo consectetur tempus vel et sapien. Curabitur enim quam, sollicitudin id iaculis id, congue euismod diam. Sed in eros nec urna lacinia porttitor ut vitae nulla. Ut mattis, erat et laoreet feugiat, lacus urna hendrerit nisi, at tincidunt dui justo at felis. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos himenaeos. Ut iaculis euismod magna et consequat. Mauris eu augue in ipsum elementum dictum. Sed accumsan, velit vel vehicula dignissim, nibh tellus consequat metus, vel fringilla neque dolor in dolor. Aliquam ac justo ut lectus iaculis pharetra vitae sed turpis. Aliquam pulvinar lorem vel ipsum auctor et hendrerit nisl molestie. Donec id felis nec ante placerat vehicula. Sed lacus risus, aliquet vel facilisis eu, placerat vitae augue.

# Índice general


1.. Introducción . . . . . 1
1.1. Teoremas . . . . . 2
1.2. Asistentes de demostraciones . . . . . 2
1.3. Arquitectura de PPA . . . . . 2
1.4. Lógica de primer orden . . . . . 3

2.. Deducción natural . . . . . 4
2.1. El sistema de deducción natural . . . . . 5
2.1.1. Reglas de inferencia . . . . . 7
2.1.2. Ejemplo introductorio . . . . . 7
2.2. Intuición detrás de las reglas . . . . . 8
2.2.1. Reglas base . . . . . 8
2.2.2. Reglas de conjunciones y disyunciones . . . . . 9
2.2.3. Reglas de implicación y negación . . . . . 9
2.2.4. Reglas de cuantificadores . . . . . 10
2.3. Ajustes para generación de demostraciones . . . . . 11
2.3.1. Hipótesis etiquetadas . . . . . 11
2.3.2. Variables libres en contexto . . . . . 11
2.4. Reglas admisibles . . . . . 12
2.5. Algoritmos . . . . . 12
2.5.1. Chequeador . . . . . 12
2.5.2. Alpha equivalencia . . . . . 13
2.5.3. Sustitución sin capturas . . . . . 13
2.5.4. Variables libres . . . . . 14

3.. El lenguaje PPA . . . . . 16
3.1. Interfaz . . . . . 21
3.1.1. Identificadores . . . . . 21
3.1.2. Comentarios . . . . . 22
3.1.3. Fórmulas . . . . . 22
3.2. Demostraciones . . . . . 22
3.2.1. Contexto . . . . . 23
3.2.2. by - mecanismo principal de demostración . . . . . 23
3.2.3. Comandos y reglas de inferencia . . . . . 24
3.2.4. Descarga de conjunciones . . . . . 26
3.2.5. Otros comandos . . . . . 27

4.. El certificador de PPA . . . . . 29
4.1. Certificados . . . . . 30
4.2. Certificador . . . . . 30
4.3. Funcionamiento del by . . . . . 32
4.3.1. Razonamiento por el absurdo . . . . . 33
4.3.2. DNF . . . . . 33

4.3.3. Contradicciones . . . 33
4.3.4. Eliminación de universales . . . 33
4.3.5. Juntando demostraciones . . . 33
4.3.6. Poder expresivo . . . 33
4.3.7. Unificación . . . 34
4.4. Comandos correspondientes a reglas de inferencia . . . 34
4.5. Descarga de conjunciones . . . 34
4.6. Comandos adicionales . . . 34

5.. Extracción de testigos de existenciales . . . 35
5.1. Lógica intuicionista . . . 37
5.2. Traducción de Friedman . . . 38
5.2.1. Traducción de doble negación . . . 38
5.2.2. El truco de Friedman . . . 38

6.. La herramienta `ppa` . . . 40
6.1. Compiladores . . . 43

7.. Conclusiones . . . 44

# 1. INTRODUCCIÓN

En este trabajo implementamos una asistente de demostración "PPA" (*Pani's proof assistant*) inspirado en Mizar.

PPA se puede referir a varias cosas: el lenguaje para escribir demostraciones, la herramienta (junto con su extracción de testigos).

## 1.1. Teoremas

In mathematics and formal logic, a theorem is a statement that has been proven, or can be proven.[a][2][3] The proof of a theorem is a logical argument that uses the inference rules of a deductive system to establish that the theorem is a logical consequence of the axioms and previously proved theorems.

In mainstream mathematics, the axioms and the inference rules are commonly left implicit, and, in this case, they are almost always those of Zermelo–Fraenkel set theory with the axiom of choice (ZFC), or of a less powerful theory, such as Peano arithmetic.[b] Generally, an assertion that is explicitly called a theorem is a proved result that is not an immediate consequence of other known theorems. Moreover, many authors qualify as theorems only the most important results, and use the terms lemma, proposition and corollary for less important theorems.

Completar con https://en.wikipedia.org/wiki/Theorem

## 1.2. Asistentes de demostraciones

- Son programs que asisten al usuario a la hora de escribir una demostración. Permiten representarlas en un programa.
- Aplicaciones: formalización de teoremas, verificación formal de programas, etc.
- Ejemplos: Coq, isabelle (isar), Mizar
- Reseña histórica de Mizar
- Ventajas: colaboración a gran escala (confianza en el checker), chequear el output de los LLMs

## 1.3. Arquitectura de PPA

- Por qué certificados (criterio de De Brujin)
- Certificados están en deducción natural. Sistema lógico que permite construir demostraciones mediante reglas de inferencia.
- PPA es un lenguaje que genera demostraciones "de bajo nivel"ND.
- razón de ser: Más práctico que demos de bajo nivel
- Implementado en Haskell

Principalmente hace dos cosas: certificar y extraer testigos.

## 1.4. Lógica de primer orden

**Def. 1.** Términos Un término puede ser

- Una variable $x$
- Una función $f(t_1, \ldots, t_n)$ donde $t_1, \ldots, t_n$ son términos

**Def. 2.** Fórmulas Una fórmula puede ser

- Un predicado $p(t_1, \ldots, t_n)$ donde $t_1, \ldots, t_n$ son términos
- $\bot$, $\top$
- Sean $A, B$ fórmulas
  - Una conjunción $A \wedge B$
  - Una disyunción $A \vee B$
  - Una implicación $A \to B$
  - Una negación $\neg A$
  - Un cuantificador universal $\forall x.A$
  - Un cuantificador existencial $\exists x.A$

(TODO: Definiciones, repaso, lo necesario)

# 2. DEDUCCIÓN NATURAL

Vamos a comenzar por las fundaciones: Queremos armar un programa que permita escribir teoremas y demostraciones. ¿Cómo se representa una demostración en la computadora? Es necesaria una representación precisa y rigurosa.

En el área de estudio de *proof theory*, en la cuál las demostraciones son tratadas como objetos matemáticos formales, nos encontramos con los *proof calculi* o *proof systems*, que son sistemas lógicos formales que permiten demostrar sentencias. Pueden ser modelados como un tipo abstracto de datos, así siendo representados en la computadora.

Por ejemplo, supongamos que tenemos la siguiente *teoría* de exámenes en la facultad, que vamos a ir iterando a lo largo de la tesis. Por ahora, en su versión proposicional. Si un alumno reprueba un final, entonces recursa. Si un alumno falta, entonces reprueba. Con estas dos, podríamos demostrar que si un alumno falta a un final, entonces recursa. Veamos cómo podría ser una demostración en lenguaje natural.

**Ejemplo 1.** Si ((reprueba entonces recursa) y (falta entonces reprueba)) y falta, entonces recursa.

Demostración:

- Asumo que falta. Quiero ver que recursa.
- Sabemos que si falta, entonces reprueba. Reprobó.
- Sabemos que si reprueba, entonces recursa.
- $\therefore$ recursó.

□

¿Cómo podría ser formalizada en un *proof system*?

## 2.1. El sistema de deducción natural

Los *proof systems* en general están compuestos por

- **Lenguaje formal**: el conjunto $L$ de fórmulas admitidas por el sistema. En nuestro caso, lógica de primer orden.
- **Reglas de inferencia**: lista de reglas que se usan para probar teoremas 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. Por ejemplo, como estamos en lógica clásica, vale el axioma *LEM* (Law of Excluded Middle): $A \vee \neg A$

El sistema particular que usamos se conoce como **deducción natural**, introducido por Gerhard Gentzen en [Gen35] (TODO: Chequear cita). Tiene dos tipos de *reglas de inferencia* para cada operador ($\wedge$, $\vee$, $\exists$, ...), que nos permiten razonar

- **Introducción**: ¿Cómo demuestro este operador?
- **Eliminación**: ¿Cómo uso este operador para demostrar otra fórmula?

Introducimos algunas definiciones preliminares, luego vemos las reglas de inferencia, un ejemplo de una demostración, y finalmente explicamos cada regla.

**Def. 3.** Contexto de demostración.

- Definimos los **contextos de demostración** $\Gamma$ como un conjunto de fórmulas, compuesto por las hipótesis que se asumen a lo largo de una demostración.
- Para algunas reglas es necesario conocer las variables libres de un contexto, que se definen de la forma usual:

$$fv(\Gamma) = \bigcup_{A \in \Gamma} fv(A)$$

**Def. 4.** Relación $\vdash$. Las reglas de inferencia de la Figura 2.1 definen la siguiente relación, que intuitivamente puede ser interpretada como "$A$ es una consecuencia de las suposiciones de $\Gamma$"

$$\Gamma \vdash A$$

**Def. 5.** Sustitución. Notamos la **sustitución sin capturas** de todas las ocurrencias libres de la variable $x$ por el término $t$ en la fórmula $A$ como

$$A\{x := t\}$$

Se explora en más detalle en la Subsección 2.5.3 Sustitución sin capturas

### 2.1.1. Reglas de inferencia

$$\frac{\Gamma \vdash \bot}{\Gamma \vdash A}\,\mathrm{E}\bot \qquad \frac{}{\Gamma \vdash \top}\,\mathrm{I}\top$$

$$\frac{}{\Gamma \vdash A \vee \neg A}\,\mathrm{LEM} \qquad \frac{}{\Gamma, A \vdash A}\,\mathrm{Ax}$$

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

$$\frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A}\,\mathrm{E}\wedge_1 \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash B}\,\mathrm{E}\wedge_2$$

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

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

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

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

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

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

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

*Fig. 2.1:* Reglas de inferencia para deducción natural de lógica de primer orden

### 2.1.2. Ejemplo introductorio

**Ejemplo 2.** Demostración de Ejemplo 1 en deducción natural. Como es en su versión proposicional, vamos a modelarlo para un solo alumno y materia. Notamos

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

Queremos probar entonces

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

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \vdash (X \to R) \wedge (F \to X)}\,\text{Ax}}{\Gamma \vdash X \to R}\,\text{E}\wedge_1 \quad \dfrac{\dfrac{\dfrac{}{\Gamma \vdash (X \to R) \wedge (F \to X)}\,\text{Ax}}{\Gamma \vdash F \to X}\,\text{E}\wedge_2 \quad \dfrac{}{\Gamma \vdash F}\,\text{Ax}}{\Gamma \vdash X}\,\text{E}\to}{\dfrac{\Gamma = (X \to R) \wedge (F \to X),\ F \vdash R}{(X \to R) \wedge (F \to X) \vdash F \to R}\,\text{I}\to}\,\text{I}\to}{\vdash \Big((X \to R) \wedge (F \to X)\Big) \to (F \to R)}$$

*Fig. 2.2:* Demostración de $\big((X \to R) \wedge (F \to X)\big) \to (F \to R)$ en deducción natural

Las demostraciones en deducción natural son un árbol, en el que cada juicio está justificado por una regla de inferencia, que puede tener sub-árboles de demostración. La raíz es la fórmula a demostrar. Paso por paso,

- I→: *introducimos* la implicación. Para demostrarla, asumimos el antecedente y en base a eso demostramos el consecuente. Es decir asumimos $(X \to R) \wedge (F \to X)$, y en base a eso queremos deducir $F \to R$.
- I→: Asumimos $F$, nos queda probar $R$. Renombramos el *contexto* de hipótesis como $\Gamma$.
- La estrategia para probar $R$ es usando la siguiente cadena de implicaciones: $F \to X \to R$, y sabemos que vale $F$. Como tenemos que probar $R$, arrancamos de atrás para adelante.
- E→: *eliminamos* una implicación, la usamos para deducir su conclusión demostrando el antecedente. Esta regla de inferencia tiene dos partes, probar la implicación $(X \to R)$, y probar el antecedente $(X)$.
  - Para probar la implicación, tenemos que usar la hipótesis *eliminando* la conjunción y especificando cuál de las dos cláusulas estamos usando.
  - Para probar el antecedente $X$, es un proceso análogo pero usando la otra implicación y el hecho de que vale $F$ por hipótesis.
- Las hojas del árbol, los casos base, suelen ser aplicaciones de la regla de inferencia Ax, que permite deducir fórmulas citando hipótesis del contexto.

## 2.2. Intuición detrás de las reglas

A continuación se explican brevemente las reglas de inferencia listadas en Figura 2.1.

### 2.2.1. Reglas base

$$\dfrac{\Gamma \vdash \bot}{\Gamma \vdash A}\,\text{E}\bot \qquad \dfrac{}{\Gamma \vdash \top}\,\text{I}\top$$

$$\dfrac{}{\Gamma \vdash A \vee \neg A}\,\text{LEM} \qquad \dfrac{}{\Gamma, A \vdash A}\,\text{Ax}$$

- E⊥: A partir de $\bot$, algo que es falso, vamos a poder deducir cualquier fórmula.

- I$\top$: $\top$ trivialmente vale siempre
- LEM: El *principio del tercero excluido* que vale en lógica clásica. Incluir este axioma es lo que hace que este sistema sea clásico.
- Ax: Como ya vimos en el Ejemplo 2 , lo usamos para deducir fórmulas que ya tenemos como hipótesis.

### 2.2.2. Reglas de conjunciones y disyunciones

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

$$\frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A}\, \mathrm{E}\wedge_1 \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash B}\, \mathrm{E}\wedge_2$$

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

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

- I$\wedge$: Para demostrar una conjunción, debemos demostrar ambas fórmulas.
- E$\wedge_1$/ E$\wedge_2$: A partir de una conjunción podemos deducir cualquiera de las dos fórmulas que la componen, porque ambas valen. Se modela con dos reglas.
- I$\vee_1$/ I$\vee_2$: Para demostrar una disyunción, alcanza con demostrar una de sus dos fórmulas. Se modela con dos reglas al igual que la eliminación de conjunción.
- E$\vee$: Nos permite deducir una conclusión a partir de una disyunción dando sub demostraciones que muestran que sin importar cual de las dos valga, asumiéndolas por separado, se puede demostrar.

### 2.2.3. Reglas de implicación y negación

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

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

- I$\to$: Para demostrar una implicación, asumimos el antecedente (agregándolo a las hipótesis) y en base a eso se demuestra el consecuente.
- E$\to$: también conocida como *modus ponens*. A partir de una implicación, si podemos demostrar su antecedente, entonces vale su consecuente.
- I$\neg$: Para demostrar una negación, lo hacemos por el absurdo: asumimos que vale la fórmula y llegamos a una contradicción. Esta regla también se suele llamar *reducción al absurdo* o RAA.
- E$\neg$: Podemos concluir un absurdo demostrando que vale una fórmula y su negación.

### 2.2.4. Reglas de cuantificadores

Las reglas de $\forall$ y $\exists$ se pueden ver como extensiones a las de $\wedge$ y $\vee$. Un $\forall$ se puede pensar como una conjunción con un elemento por cada uno del dominio sobre el cual se cuantifica, y análogamente un $\exists$ como una disyunción.

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

- I$\forall$: Para demostrar un $\forall x.A$, quiero ver que sin importar el valor que tome $x$ yo puedo demostrar $A$. Pero para eso en el contexto $\Gamma$ no tengo que tenerlo ligado a nada, sino no lo estaría demostrando en general.
- E$\forall$: Para usar un $\forall x.A$ para demostrar, como vale para todo $x$, puedo instanciarlo en *cualquier término* $t$.

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

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

- I$\exists$: Para demostrar un $\exists$, alcanza con instanciar $x$ en un término $t$ para el que sea cierto.
- E$\exists$: Para usar un $\exists$ para demostrar, es parecido a E$\vee$. Como tenemos que ver que vale para cualquier $x$, podemos concluir $B$ tomando como hipótesis $A$ con $x$ sin instanciar.

**Ejemplo 3.** Para ejemplificar el uso de las reglas de cuantificadores, extendemos el Ejemplo 2 a primer orden. Usamos

- $a$ representa un alumno, $m$ una materia y $e$ un examen.
- $X(a, e) \equiv \text{reprueba}(a, e)$
- $R(a, m) \equiv \text{recursa}(a, m)$
- $F(a, e) \equiv \text{falta}(a, e)$

Vamos a tomar los siguientes como *axiomas*, que van a formar parte del contexto inicial de la demostración. Es lo mismo que haremos en PPA para modelar teorías de primer orden.

- Si un alumno reprueba el final de una materia, entonces recursa

$$\forall a.\ \forall m.\ (X(a, \text{final}(m)) \to R(a, m))$$

- Si un alumno falta a un examen, lo reprueba

$$\forall a.\ \forall e.\ (F(a, e) \to X(a, e))$$

Definimos

$$\Gamma_0 = \{\forall a.\ \forall m.\ X(a, \text{final}(m)) \to R(a, m), \forall a.\ \forall e.\ F(a, e) \to X(a, e)\}$$

Luego, queremos probar $\Gamma_0 \vdash \forall a.\ \forall m.\ F(a, \text{final}(m)) \to R(a, m)$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma_1 \vdash \forall a \forall m.\ X(a, \text{final}(m)) \to R(a,m)}\,\text{Ax}}{\Gamma_1 \vdash \forall m.\ X(a, \text{final}(m)) \to R(a,m)}\,\text{E}\forall}{\Gamma_1 \vdash X(a, \text{final}(m)) \to R(a,m)}\,\text{E}\forall \qquad \Pi}{\dfrac{\dfrac{\dfrac{\Gamma_1 = \Gamma_0, F(a, \text{final}(m)) \vdash R(a,m)}{\Gamma_0 \vdash F(a, \text{final}(m)) \to R(a,m)}\,\text{I}\to}{\Gamma_0 \vdash \forall m.\ F(a, \text{final}(m)) \to R(a,m)}\,\text{I}\forall}{\Gamma_0 \vdash \forall a.\ \forall m.\ F(a, \text{final}(m)) \to R(a,m)}\,\text{I}\forall}\,\text{E}\to \qquad \Gamma_1 \vdash X(a, \text{final}(m))$$

Con

$$\Pi = \dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma_1 \vdash \forall a.\ \forall e.\ F(a,e) \to X(a,e)}\,\text{Ax}}{\Gamma_1 \vdash \forall e.\ F(a,e) \to X(a,e)}\,\text{E}\forall}{\Gamma_1 \vdash F(a, \text{final}(m)) \to X(a, \text{final}(m))}\,\text{E}\forall \qquad \dfrac{}{\Gamma_1 \vdash F(a, \text{final}(m))}\,\text{Ax}}{\Gamma_1 \vdash X(a, \text{final}(m))}\,\text{E}\to$$

*Fig. 2.3:* Demostración con cuantificadores en deducción natural

## 2.3. Ajustes para generación de demostraciones

### 2.3.1. Hipótesis etiquetadas

En las secciones anteriores presentamos a los contextos $\Gamma$ como *conjuntos* de fórmulas. Pero en realidad, para mayor claridad en las demostraciones, vamos a querer que las hipótesis estén **etiquetadas**. Para permitirlo, las diferencias son las siguientes.

- Los contextos $\Gamma$ son conjuntos de pares $h : A$ de etiquetas y fórmulas.
- Las reglas que hacen uso de hipótesis, lo hacen nombrándolas.
  (TODO: E$\exists$)

$$\dfrac{\boldsymbol{h} : A \in \Gamma}{\Gamma \vdash A}\,\text{Ax} \qquad \dfrac{\Gamma, \boldsymbol{h} : A \vdash B}{\Gamma \vdash A \to B}\,\text{I}\to \qquad \dfrac{\Gamma, \boldsymbol{h} : A \vdash \bot}{\Gamma \vdash \neg A}\,\text{I}\neg$$

(DUDA: No veo por qué con esta presentación sería necesario tener a los nombres de las reglas con las etiquetas, por ej. I$\to_h$)

### 2.3.2. Variables libres en contexto

Las reglas I$\forall$ y E$\exists$ requieren que la variable del cuantificador no esté libre en el contexto. Esto representó un problema en la generación de demostraciones. Por ejemplo cuando se citan otros teoremas y se insertan sus demostraciones, si usan las mismas variables, lo cual es usual, llevaban a conflictos. Para evitar esos problemas, lo cambiamos por algo más permisivo pero que mantiene validez: en lugar de fallar, remueve del contexto todas las hipótesis que contengan libre esa variable en la sub-demostración.

$$\dfrac{\hat{\boldsymbol{\Gamma}} \vdash A \qquad x \notin fv(\Gamma)}{\Gamma \vdash \forall x.A}\,\text{I}\forall$$

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

donde

$$\hat{\Gamma} = \{A \in \Gamma \mid x \notin fv(A)\}$$

## 2.4. Reglas admisibles

Antes mencionamos *modus tollens* como regla de inferencia, pero no aparece en la Figura 2.1. Esto es porque nos va a interesar tener un sistema lógico minimal: no vamos a agregar reglas de inferencia que se puedan deducir a partir de otras, es decir, *reglas admisibles*. Nos va a servir para simplificar el resto de PPA, dado que vamos a generar demostraciones en deducción natural y operar sobre ellas. Mientras más sencillas sean las partes con las que se componen, mejor. Las reglas admisibles las podemos demostrar para cualquier fórmula, así luego podemos usarlas como *macros*.

**Ejemplo 4.** *Modus tollens*

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

(NOTA: cut y dnegElim las voy a contar en la sección del certifier, que es donde se usan.)

## 2.5. Algoritmos

A continuación describimos los algoritmos que son necesarios para la implementación de deducción natural. El chequeo de las demostraciones, alpha equivalencia de fórmulas, sustitución sin capturas, y variables libres

### 2.5.1. Chequeador

Sección reescrita!

El algoritmo de chequeo de una demostración en deducción natural consiste en recorrer recursivamente el árbol de demostración, chequeando que todas las inferencias sean válidas y manteniendo un contexto $\Gamma$ en el camino. Se chequea Que se usan para demostrar la fórmula que corresponde (no un I$\wedge$ para un $\vee$) y que cumplen con las condiciones impuestas.

El módulo que se encarga de implementarlo es el `Checker`, su función principal es

```
check :: Env -> Proof -> Form
```

donde **`Env`** es el contexto $\Gamma$, **`Proof`** es el término de demostración en deducción natural y **`Form`** es la fórmula que demuestra.

### 2.5.2. Alpha equivalencia

Si tenemos una hipótesis $\exists x.f(x)$, sería ideal poder usarla para demostrar a partir de ella una fórmula $\exists y.f(y)$. Si bien no son exactamente iguales, son **alpha-equivalentes**: su estructura es la misma, pero tienen nombres diferentes para variables *ligadas* (no libres)

**Def. 6.** Alpha equivalencia. Se define la relación $\overset{\alpha}{=}$ como la que permite renombrar variables ligadas evitando capturas. Es la congruencia más chica que cumple con

$$(\forall x.A) \overset{\alpha}{=} (\forall y.A') \iff A\{x := z\} \overset{\alpha}{=} A'\{y := z\} \text{ con } z \text{ fresca}$$
$$(\exists x.A) \overset{\alpha}{=} (\exists y.A') \iff A\{x := z\} \overset{\alpha}{=} A'\{y := z\} \text{ con } z \text{ fresca}$$

Para implementarlo, un algoritmo naïve podría ser cuadrático: chequeamos recursivamente la igualdad estructural de ambas fórmulas. Si nos encontramos con un cuantificador con variables con nombres distintos, digamos $x$ e $y$, elegimos una nueva variable *fresca* (para evitar capturas) y lo renombramos recursivamente en ambos. Luego continuamos con el algoritmo. Si en la base nos encontramos con dos variables, tienen que ser iguales.

Para hacerlo un poco más eficiente, se implementó un algoritmo lineal en la estructura de la fórmula. Mantenemos dos sustituciones de variables, una para cada fórmula. Si nos encontramos con $\exists x.f(x)$ y $\exists y.f(y)$, vamos a elegir una variable fresca igual que antes (por ejemplo $z$), pero en vez de renombrar recursivamente, que lo hace cuadrático, insertamos en cada sustitución los renombres $x \mapsto z$ y $y \mapsto z$. Luego, cuando estemos comparando dos variables libres, chequeamos que *sus renombres* sean iguales. En este ejemplo son alpha equivalentes, pues

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

### 2.5.3. Sustitución sin capturas

Notamos la sustitución de todas las ocurrencias libres de la variable $x$ por un término $t$ en una fórmula $A$ como $A\{x := t\}$. Esto se usa en algunas reglas de inferencia,

$$\frac{\Gamma \vdash \forall x.A}{\Gamma \vdash A\{x := t\}} \text{E}\forall$$

Pero queremos evitar **captura de variables**. Por ejemplo, en

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

si sustituimos sin más, estaríamos involuntariamente "capturando" a $y$. Si hiciéramos que falle, tener que escribir las demostraciones con estos cuidados puede ser muy frágil y propenso a errores, por lo que es deseable que se resuelva *automáticamente*: cuando nos encontramos con una captura, sustituimos la variable ligada de forma que no ocurra.

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

donde $z$ es una variable *fresca*.

**Def. 7.** Sustitución sin capturas. Se define inductivamente en la estructura la fórmula. Sean $y$ una variable y $t$ un término cualquiera.

- Términos

$$x\{y := t\} = \begin{cases} t & \text{si } x = y \\ x & \text{si no} \end{cases}$$
$$f(t_1, \ldots, t_n)\{y := t\} = f(t_1\{y := t\}, \ldots, t_n\{y := t\})$$

- Fórmulas

$$\begin{aligned}
\bot\{y := t\} &= \bot \\
\top\{y := t\} &= \top \\
p(t_1, \ldots, t_n)\{y := t\} &= p(t_1\{y := t\}, \ldots, t_n\{y := t\}) \\
(A \wedge B)\{y := t\} &= A\{y := t\} \wedge B\{y := t\} \\
(A \vee B)\{y := t\} &= A\{y := t\} \vee B\{y := t\} \\
(A \to B)\{y := t\} &= A\{y := t\} \to B\{y := t\} \\
(\neg A)\{y := t\} &= \neg A\{y := t\} \\
(\forall x.A)\{y := t\} &= \begin{cases} \forall z.(A\{x := z\})\{y := t\} & \text{si } x = y, \text{ con } z \text{ fresca} \\ \forall x.A\{y := t\} & \text{si no} \end{cases} \\
(\exists x.A)\{y := t\} &= \begin{cases} \exists z.(A\{x := z\})\{y := t\} & \text{si } x = y, \text{ con } z \text{ fresca} \\ \exists x.A\{y := t\} & \text{si no} \end{cases}
\end{aligned}$$

Para implementarlo, cada vez que nos encontramos con una captura, vamos a *renombrar* la variable del cuantificador por una nueva, fresca. Al igual que la alpha igualdad, esto se puede implementar de forma naïve cuadrática pero lo hicimos lineal. Mantenemos un único mapeo a lo largo de la sustitución, y cada vez que nos encontramos con una variable libre, si son iguales la sustituimos por el término, y si está mapeada la renombramos.

### 2.5.4. Variables libres

(DUDA: Mover a la sección de LPO en la introducción? esto es well-known)

**Def. 8.** Variables libres de fórmulas y términos. Se definen inductivamente en su estructura de la siguiente forma.

- Términos

$$fv(x) = \{x\}$$
$$fv(f(t_1, \ldots, t_n)) = \bigcup_{i \in 1 \ldots n} fv(t_i)$$

- Fórmulas

$$
\begin{aligned}
fv(\bot) &= \emptyset \\
fv(\top) &= \emptyset \\
fv(p(t_1, \ldots, t_n)) &= \bigcup_{i \in 1 \ldots n} fv(t_i) \\
fv(A \wedge B) &= fv(A) \cup fv(B) \\
fv(A \vee B) &= fv(A) \cup fv(B) \\
fv(A \to B) &= fv(A) \cup fv(B) \\
fv(\neg A) &= fv(A) \\
fv(\forall x.A) &= fv(A) \setminus x \\
fv(\exists x.A) &= fv(A) \setminus x
\end{aligned}
$$

**Def. 9.** Variables libres de una demostración. Sea $\Pi$ una demostración. $fv(\Pi)$ son las variables libres de todas las fórmulas que la componen. Por ejemplo, para la siguiente

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

se tiene $fv(\Pi) = fv(A) \cup fv(B)$

# 3. EL LENGUAJE PPA

- Interfaz de PPA. Acá tienen que quedar claras todas las intuiciones desde el punto de vista de un usuario. Mencionar que es un buen momento para que vayan y prueben el programa (comando `check` nada más) ✓
  - Programas, teoremas, demostraciones como listas de pasos que reducen la tesis hasta agotarla. ✓
  - Comandos 1 por 1. Similar al README que ya existe pero más facha ✓
  - Ejemplos de demostraciones. ✓
- Certificador: componente de PPA que çertifica"las demostraciones, generando un certificado en deducción natural. Implicó escribir muchos meta-teoremas.
  - Formalización de muchos teoremas y axiomas: contextos (vale en el prefijo)
  - Proof y proof steps, simplificación de la interfaz y mapeo de comandos a steps
  - Implementación de cada comando
  - By y solver para resolver varios. DNF. Extensión con foralls consecutivos. Demostración / justificación de que es correcto y completo para LP, pero heurístico para LPO (mostrar un caso en el que no funcione)
  - Descarga de conjunciones
  - Uso de dneg elim como razonamiento por el absurdo para demostrar deMorgan y equivalencias.

El lenguaje PPA (*Pani's Proof Assistant*) se construye sobre las fundaciones de deducción natural. Es un asistente de demostración que permite escribir de una forma práctica demostraciones de cualquier teoría de lógica clásica de primer orden. En este capítulo nos vamos a centrar en el lenguaje, la interfaz del programa `ppa`. Los detalles teóricos de cómo está implementado son contados en el Capítulo 4, y los detalles de implementación y su instalación en el Capítulo 6. Para introducirlo veamos un ejemplo: representamos el mismo de alumnos del Ejemplo 3 pero con un poco más de sofisticación.

```
1  axiom reprueba_recu_parcial_recursa: forall A. forall M.
2     (reprueba(A, parcial(M)) & reprueba(A, recu(parcial(M))))
3        -> recursa(A, M)
4
5  axiom rinde_recu_reprobo_parcial: forall A. forall P.
6     rinde(A, recu(P)) -> reprueba(A, P)
7
8  axiom reprobo_rinde: forall A. forall P.
9     reprueba(A, P) -> rinde(A, P)
10
11 axiom falta_reprueba: forall A. forall P.
12    falta(A, P) -> reprueba(A, P)
13
14 theorem reprueba_recu_recursa:
15    forall A. forall M.
16       reprueba(A, recu(parcial(M))) -> recursa(A, M)
17 proof
18    let A
19    let M
20    suppose reprueba_recu: reprueba(A, recu(parcial(M)))
21
22    claim reprueba_p: reprueba(A, parcial(M))
23    proof
24       have rinde_recu: rinde(A, recu(parcial(M)))
25          by reprueba_recu, reprobo_rinde
26
27       hence reprueba(A, parcial(M))
28          by rinde_recu_reprobo_parcial
29    end
30
31    hence recursa(A, M)
32       by reprueba_recu,
33          reprueba_p,
34          reprueba_recu_parcial_recursa
35 end
36
37 theorem falta_recu_recursa:
38    forall A. forall M.
39       falta(A, recu(parcial(M))) -> recursa(A, M)
40 proof
41    let A'
42    let M'
43
44    suppose falta_recu: falta(A', recu(parcial(M')))
45
46    have reprueba_recu: reprueba(A', recu(parcial(M')))
47       by falta_recu, falta_reprueba
48
49    hence recursa(A', M') by reprueba_recu_recursa
50 end
```

*Fig. 3.1:* Programa de ejemplo completo en PPA. Demostraciones de alumnos y parciales.

La primer parte de todo programa en PPA es definir los axiomas de la *teoría de primer orden* con la que se está trabajando. Como no se chequean tipos, no es necesario definir explícitamente los símbolos de predicados y de función. Pero se agregan a modo informativo como un comentario. Son fórmulas que son siempre consideradas como válidas. Definimos los siguientes,

- `reprueba_recu_parcial_recursa`: Si un alumno reprueba el parcial y el recuperatorio de una materia, la recursa.
- `reprobo_rinde`: Si un alumno reprobó un examen, es porque lo rindió.
- `rinde_recu_reprobo_parcial`: Si un alumno rinde el recu de un parcial, es porque reprobó la primer instancia.
- `falta_reprueba`: Si un alumno falta a un exámen, lo reprueba.

```
1  /* Teoría de alumnos y exámenes
2
3  Predicados
4     - reprueba(A, P): El alumno A reprueba el parcial P
5     - recursa(A, M): El alumno A recursa la materia M
6
7  Funciones
8     - parcial(M): El parcial de una materia
9     - recu(P): El recuperatorio de un parcial
10 */
11
12 axiom reprueba_recu_parcial_recursa: forall A. forall M.
13    (reprueba(A, parcial(M)) & reprueba(A, recu(parcial(M))))
14       -> recursa(A, M)
15
16 axiom rinde_recu_reprobo_parcial: forall A. forall P.
17    rinde(A, recu(P)) -> reprueba(A, P)
18
19 axiom reprobo_rinde: forall A. forall P.
20    reprueba(A, P) -> rinde(A, P)
21
22 axiom falta_reprueba: forall A. forall P.
23    falta(A, P) -> reprueba(A, P)
```

En base a eso demostramos dos teoremas. El primero, `reprueba_recu_recursa`, nos permite concluir que un alumno recursa solo a partir de que reprueba el recuperatorio. Esto es porque a partir de esto, con el resto de los axiomas, podemos deducir que también reprobó el parcial: si reprueba el recuperatorio es porque lo rindió, y si rindió el recuperatorio, es porque reprobó el parcial.

```
theorem reprueba_recu_recursa:
   forall A. forall M.
      reprueba(A, recu(parcial(M))) -> recursa(A, M)
proof
   let A
   let M
   suppose reprueba_recu: reprueba(A, recu(parcial(M)))

   claim reprueba_p: reprueba(A, parcial(M))
   proof
      have rinde_recu: rinde(A, recu(parcial(M)))
         by reprueba_recu, reprobo_rinde

      hence reprueba(A, parcial(M))
         by rinde_recu_reprobo_parcial
   end

   hence recursa(A, M)
      by reprueba_recu,
         reprueba_p,
         reprueba_recu_parcial_recursa
end
```

Para demostrar un teorema, tenemos que agotar su *tesis* reduciéndola sucesivamente con *proof steps*. Una demostración es correcta si todos los pasos son lógicamente ciertos, y luego de ejecutar todos los pasos, la tesis se reduce por completo.

- **`let`** permite demostrar un **`forall`**, asignando un nombre a la variable general, y *reduce* la tesis a su fórmula.
- **`suppose`** permite demostrar una implicación. Agrega como hipótesis al contexto el antecedente, permitiendo nombrarlo, y reduce la tesis al consecuente.
- **`claim`** permite agregar una sub-demostración, cuya fórmula se agrega como hipótesis.
- **`have`** agrega una hipótesis sin reducir la tesis.
- **`by`** es el mecanismo principal de demostración. Permite deducir fórmulas a partir de otras. Es completo para lógica proposicional, y heurístico para primer orden. Unifica las variables de los **`forall`**.
- **`thus`** permite reducir parte o la totalidad de la tesis
- **`hence`** es igual a thus, pero incluye implícitamente la hipótesis anterior a las justificaciones del **`by`**.

Finalmente, a partir del teorema anterior y el axioma `falta_reprueba` podemos demostrar que si un alumno falta a un recuperatorio, recursa la materia.

```
theorem falta_recu_recursa:
   forall A. forall M.
      falta(A, recu(parcial(M))) -> recursa(A, M)
proof
   let A'
   let M'

   suppose falta_recu: falta(A', recu(parcial(M')))

   have reprueba_recu: reprueba(A', recu(parcial(M')))
      by falta_recu, falta_reprueba

   hence recursa(A', M') by reprueba_recu_recursa
end
```

Al ejecutarlo con `ppa`, se *certifica* la demostración, generando un certificado de deducción natural, y luego se chequea que sea correcto. Si se escribió una demostración que no es lógicamente válida, el certificador reporta el error. No debería fallar nunca el chequeo sobre el certificado.

## 3.1. Interfaz

PPA es un lenguaje que permite escribir demostraciones de cualquier teoría de lógica de primer orden. Está inspirado en el *mathematical vernacular* introducido por Freek Wiedijk en [Wie]. En esta sección nos concentramos en la interfaz de usuario, sin entrar en detalle en cómo está implementada.

Un programa de PPA consiste en una lista de **declaraciones**: axiomas y teoremas, que se leen en orden el inicio hasta el final.

- Los axiomas se asumen válidos, se usan para modelar la *teoría de primer orden* sobre la cual hacer demostraciones

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

- Los teoremas deben ser demostrados, y se pueden citar todas las hipótesis previas, que consideran ciertas.

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

### 3.1.1. Identificadores

Los identificadores se dividen en tres tipos:

- **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 (`"..."`)

  `\"[^\"]*\"`

### 3.1.2. Comentarios

Se pueden dejar comentarios de una sola línea (`//`) o multilínea (`/* ... */`)

### 3.1.3. Fórmulas

Las fórmulas están compuestas por,

- **Términos**
  - **Variables**: `<var>`. Ejemplos: `_x`, `X`, `X'''`, `Alumno`.
  - **Funciones**: `<id>(<term>, ..., <term>)`. Los argumentos son opcionales, pudiendo tener funciones 0-arias (constantes). Ejemplos: `c`, `f(_x, c, X)`.
- **Predicados**: `<id>(<term>, ..., <term>)`. Los argumentos son opcionales, pudiendo tener predicados 0-arios (variables proposicionales). Ejemplos: `p(c, f(w), X)`, `A`, `<(n, m)`
- **Conectivos binarios**.
  - `<form> & <form>` (conjunción)
  - `<form> | <form>` (disyunción)
  - `<form> -> <form>` (implicación)
- **Negación** $\sim$ `<form>`
- **Cuantificadores**
  - `exists <var>. <form>`
  - `forall <var>. <form>`
- `true`, `false`
- **Paréntesis**: `( <form> )`

## 3.2. Demostraciones

Las demostraciones consisten en una lista de *proof steps* o comandos que pueden reducir sucesivamente la *tesis* (fórmula a demostrar) hasta agotarla por completo. Corresponden aproximadamente a reglas de inferencia de deducción natural.

### 3.2.1. Contexto

Las demostraciones llevan asociado un *contexto* con todas las hipótesis que fueron asumidas (como los axiomas), o demostradas: tanto teoremas anteriores como sub-teoremas y otros comandos que agregan hipótesis a él.

### 3.2.2. by - mecanismo principal de demostración

El mecanismo principal de demostración directa o *modus ponens* es el **by**, que afirma que un hecho es consecuencia de una lista de hipótesis (su "justificación"). Esto permite eliminar universales e implicaciones. Por detrás hay un solver completo para lógica proposicional pero heurístico para primer orden (elimina todos los foralls de a lo sumo una hipótesis, no intenta con más de una). Exploramos las limitaciones en (TODO: ref).

En general, `... by <justification>` se interpreta como que `...` es una consecuencia lógica de las fórmulas que corresponden a las hipótesis de `<justification>`, que deben estar declaradas anteriormente y ser parte del contexto. Ya sea por axiomas o teoremas, u otros comandos que agreguen hipótesis (como **suppose**). Puede usarse de dos formas principales, **thus** y **have**

- `thus <form> by <justification>`.

  Si `<form>` es *parte* de la tesis (ver Subsección 3.2.4 Descarga de conjunciones), y es consecuencia lógica de las justificaciones, lo demuestra automáticamente y lo descarga de la tesis.

  Por ejemplo, para eliminación de implicaciones

```
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
```

  Y para eliminación de cuantificadores universales

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

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

- `have <form> by <justification>`.

  Igual a **thus**, pero permite introducir afirmaciones *auxiliares* que no son parte de la tesis, sin reducirla, y las agrega a las hipótesis para usar luego. Por ejemplo, la demostración anterior la hicimos en un solo paso con el **thus**, pero podríamos haberla hecho en más de uno con una afirmación auxiliar intermedia.

```
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
```

Ambas tienen su contraparte con *azúcar sintáctico* que agrega automáticamente la hipótesis anterior a la justificación, a la que también se puede referir con `-`.

| Comando | Alternativo | ¿Reduce la tesis? |
|---|---|---|
| **thus** | **hence** | Si |
| **have** | **then** | No |

Por ejemplo,

```
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
```

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

theorem t1'': a -> c
proof
   suppose -: a
   then -: b by ax1
   hence c by ax2
end
```

En todos, el **by** es opcional. En caso de no especificarlo, debe ser una tautología.

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

### 3.2.3. Comandos y reglas de inferencia

Muchas reglas de inferencia de deducción natural (2.1) tienen una correspondencia directa con comandos. Como se puede ver en Tabla 3.1 Reglas de inferencia y comandos, la mayor parte del trabajo engorroso y bajo nivel de escribir demostraciones en deducción natural se lleva a un más alto nivel con el uso del **by**.

| Regla | Comando |
|---|---|
| I∃ | **take** |
| E∃ | **consider** |
| I∀ | **let** |
| E∀ | **by** |
| $I\vee_1$ | **by** |
| $I\vee_2$ | **by** |
| E∨ | **cases** |
| I∧ | **by** |
| $E\wedge_1$ | **by** |
| $E\wedge_2$ | **by** |
| I→ | **suppose** |
| E→ | **by** |
| I¬ | **suppose** |
| E¬ | **by** |
| I⊤ | **by** |
| E⊥ | **by** |
| LEM | **by** |
| Ax | **by** |

*Tab. 3.1:* Reglas de inferencia y comandos

- **suppose** (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 $\neg A \equiv A \to \bot$, se puede usar para razonar por el absurdo, tomando $B = \bot$.

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

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

- **cases** (E∨)

  Permite razonar por casos. Para cada uno se debe demostrar la tesis en su totalidad por separado.

```
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
```

```
9           thus a by right
10     end
11 end
```

No es necesario que los casos sean exactamente iguales a como están presentados en la hipótesis, solo debe valer que la disyunción de ellos sea consecuencia de ella. Es decir, para poder usar

```
cases by h1, ..., hn
   case c1
   ...
   case cm
end
```

Tiene que valer `c1 | ... | cm by h1, ..., hn`.

Por lo que en el ejemplo anterior, podríamos haber usado el mismo case incluso si la hipótesis fuera `~((~a | ~b) & (~c | ~a))`, pues es equivalente a `(a & b) | (c & a)`

- **`take`** (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∃)

  Permite razonar sobre una variable que cumpla con un existencial. Si se puede justificar **`exists`** `X. p(X)`, permite razonar sobre `X`.

  El comando **`consider`** `X` **`st`** `h: p` **`by`** `...` agrega `p` como hipótesis al contexto para el resto de la demostración. El **`by`** debe justificar **`exists`** `X. p(X)`.

  Valida que `X` no esté libre en la tesis o el contexto.

  También es posible usar una variable y fórmula $\alpha$-equivalente, por ejemplo si podemos justificar **`exists`** `X. p(X)`, podemos usarlo para **`consider`** `Y` **`st`** `h: p(Y)` **`by`** `...`

- **`let`** (I∀)

  Permite demostrar un cuantificador universal. Si se tiene **`forall`** `X. p(X)`, luego de **`let`** `X` la tesis se reduce a `p(X)` con un `X` genérico. Puede ser el mismo nombre de variable o uno diferente.

### 3.2.4. Descarga de conjunciones

Si la tesis es una conjunción, se puede probar solo una parte de ella y se reduce al resto.

*Fig. 3.2:* Descarga de conjunción 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
```

Esto puede ser prácticamente en cualquier orden.

*Fig. 3.3:* Descarga de conjunción 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
```

### 3.2.5. Otros comandos

- **equivalently**: Permite reducir la tesis a una fórmula equivalente. Útil para usar descarga de conjunciones.

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

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

- **claim**: Permite demostrar una afirmación auxiliar. Útil para ordenar las demostraciones sin tener que definir otro teorema. Ejemplo en Figura 3.1 Programa de ejemplo completo en PPA. Demostraciones de alumnos y parciales

```
theorem t: <form>
proof
   claim <name>: <form>
   proof
      <proof>
```

```
    end
end
```

# 4. EL CERTIFICADOR DE PPA

Certificador: componente de PPA que çertifica"las demostraciones, generando un certificado en deducción natural. Implicó escribir muchos meta-teoremas.

- Formalización de muchos teoremas y axiomas: contextos (vale en el prefijo)
- Proof y proof steps, simplificación de la interfaz y mapeo de comandos a steps
- Implementación de cada comando
- By y solver para resolver varios. DNF. Extensión con foralls consecutivos. Demostración / justificación de que es correcto y completo para LP, pero heurístico para LPO (mostrar un caso en el que no funcione)
- Descarga de conjunciones
- Uso de dneg elim como razonamiento por el absurdo para demostrar deMorgan y equivalencias.

En la sección pasada vimos cómo se puede usar PPA para demostrar teoremas. Pero, ¿cómo funciona por detrás? ¿Como asegura la validez lógica de las demostraciones escritas por el usuario?

## 4.1. Certificados

Los programas de PPA se **certifican**, generando una demostración en deducción natural. ¿Por qué? El lenguaje de PPA es complejo, la implementación no es trivial. Si se escribe una demostración, para confiar en que es correcta hay que confiar en la implementación de PPA.

Pero si PPA genera una demostración de bajo nivel, que usa las reglas de un sistema lógico simple como deducción natural, entonces cualquiera que desconfíe podría fácilmente escribir un chequeador, o usar uno confiable. Por eso genera demostraciones en deducción natural, haciendo que cumpla con el **criterio de de Bruijn**

**Def. 10.** Criterio de de Bruijn [BW05]. Un asistente de demostración que satisface que sus demostraciones puedan ser chequeadas por un programa independiente, pequeño y confiable se dice que cumple con el criterio de de Bruijn.

El módulo de PPA que *certifica* las demostraciones de alto nivel de PPA generando una demostración en deducción natural es el `Certifier`. Si bien toda demostración que genere debería ser correcta, para atajar posibles errores siempre se chequean con el `Checker` de DN.

(DUDA: Para pablo: pero si emite certificados que no corresponden a la demo original y chequean siempre, por ej. siempre el mismo, no estaría mal igual? Tenés que confiar también en la parte que emite el certificado.)

## 4.2. Certificador

El `Certifier`en realidad no genera una sola demostración de deducción natural, sino que al poder haber más de un teorema en un archivo PPA, este genera un **contexto** compuesto por una lista ordenada de **hipótesis**. Hay dos tipos de hipótesis

- Teoremas: Son fórmulas con demostraciones asociadas.
- Axiomas: Son fórmulas que se asumen válidas (pueden ser usadas para modelar una teoría)

```
1  axiom ax1: q
2  axiom ax2: q -> p
3  axiom ax3: p -> r
4
5  theorem t1: p
6  proof
7     thus p by ax1, ax2
8  end
9
10 theorem t2: r
11 proof
12    thus r by t1, ax3
13 end
```

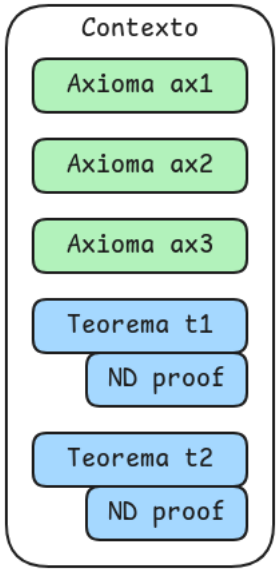


*Fig. 4.1:* Contexto resultante de certificar de un programa

Por lo tanto, en vez de chequear demostraciones, se chequean contextos: Una demostración será válida en el *prefijo estricto del contexto* que la contiene. Es decir, a la hora de chequearla, se deben asumir como ciertas todas las hipótesis que fueron definidas previamente.

Cada demostración de un teorema tendrá además un *contexto local*. Las afirmaciones auxiliares que no afectan la tesis (**`have`**, **`claim`**, **`consider`**, etc.) se agregan como teoremas. Por lo tanto, cuando se citen, se pegan sus demostraciones. Por otro lado, algunos comandos agregan axiomas, los mismos que en deducción natural agregan fórmulas al contexto (**`suppose`** y **`consider`**). Es correcto asumir como ciertas esas hipótesis, porque lo mismo se hará durante el chequeo de la demostración en deducción natural.

```
1 axiom ax1: p -> q
2 theorem t: (q -> r) -> p -> r
3 proof
4    suppose h1: (q -> r)
5    suppose h2: p
6    then tq: q by ax1
7    hence r by h1
8 end
```

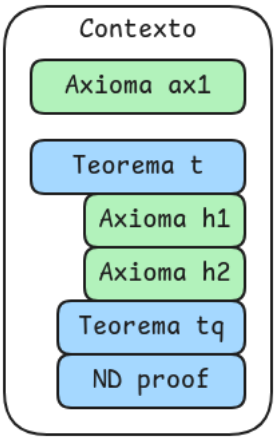


*Fig. 4.2:* Contexto local

## 4.3. Funcionamiento del by

El **by** es el mecanismo principal de demostración en PPA, y el corazón del `Certifier`. Genera **automáticamente** una demostración de que una fórmula es consecuencia lógica de una lista de hipótesis.

Sea $A$ una fórmula. Supongamos que queremos demostrar **thus** `A` **by** `h1, ..., hn` y que en el contexto tenemos que las hipótesis $h_i$ corresponden a fórmulas $B_i$, con $i \in \{1, ..., n\}, n \in \mathbb{N}$. Primero veamos la idea general de la estrategia y luego profundizamos en cada paso.

- Queremos demostrar la implicación

$$B_1 \wedge \ldots \wedge B_n \to A$$

- Razonamos por el absurdo: asumiendo la negación encontramos una contradicción

$$\begin{aligned} \neg(B_1 \wedge \ldots \wedge B_n \to A) &\equiv \neg(\neg(B_1 \wedge \ldots \wedge B_n) \vee A) \\ &\equiv B_1 \wedge \ldots \wedge B_n \wedge \neg A \end{aligned}$$

- Convertimos la fórmula a forma normal disyuntiva (DNF), disyunciones de conjunciones de literales. Un literal será un predicado, una negación de un literal o una fórmula con cuantificadores.

$$(a_1^1 \wedge \ldots \wedge a_{n_1}^1) \vee \ldots \vee (a_1^m \wedge \ldots \wedge a_{n_m}^m)$$

  donde $m \in \mathbb{N}$ es el número de cláusulas, $n_1, \ldots, n_m \in \mathbb{N}$ es la cantidad de la cantidad de fórmulas de cada cláusula y $a_j^i$ es la $j$-ésima fórmula de la $i$-ésima cláusula.

- Buscamos una contradicción refutando cada cláusula (conjunción de literales) individualmente. Una cláusula $a_1 \wedge \ldots \wedge a_n$ será refutable si
  - Contiene $\bot$

- Contiene dos fórmulas opuestas $(a, \neg a)$
- Eliminando existenciales consecutivos y re-convirtiendo a DNF, se consigue una refutación $(\neg p(k), \forall x.\ p(x))$

La complejidad del mecanismo no reside solo en tener que realizar todos estos pasos. El desafío principal fue **generar la demostración en deducción natural**.

**Ejemplo 5.** Ejemplo sin cuantificadores Veamos un ejemplo proposicional. (TODO: ejemplo)

**Ejemplo 6.** Ejemplo con cuantificadores Veamos un ejemplo con cuantificadores.

### 4.3.1. Razonamiento por el absurdo

(TODO: DnegElim como regla admisible, y cómo permite razonar por el absurdo)

### 4.3.2. DNF

(TODO: DNF)

### 4.3.3. Contradicciones

(TODO: contradicciones)

### 4.3.4. Eliminación de universales

### 4.3.5. Juntando demostraciones

(TODO: Cut como regla admisible)

### 4.3.6. Poder expresivo

(DUDA: Está bien decir poder expresivo?)
El solver implementado es **completo para lógica proposicional**, pero heurístico para lógica de primer orden. Esto es aceptable, puesto que la satisfacibilidad de lógica de primer orden es indecidible (TODO: Cita?), y además el objetivo del trabajo no fue dar el mejor solver, sino dar alguno.
(TODO: Citar Hilbert 1950)

**Teorema 1.** El solver es completo para lógica proposicional. (TODO: Demo)

Limitaciones

No puede demostrar por ej.
(TODO: ejemplo elim 2 foralls)
(TODO: otro ejemplo más burdamente complejo que requiera sat sovler)

### 4.3.7. Unificación

## 4.4. Comandos correspondientes a reglas de inferencia

Como se puede apreciar en Tabla 3.1 Reglas de inferencia y comandos, la mayoría de los comandos se corresponden directamente con reglas de inferencia, por lo que su traducción es más sencilla.

(TODO: trad)

## 4.5. Descarga de conjunciones

## 4.6. Comandos adicionales

(TODO: add)

# 5. EXTRACCIÓN DE TESTIGOS DE EXISTENCIALES

Puntos a abordar
- mencionar realizabilidad clásica. Related work capaz en la conclusión - hacer una investigación de otras formas de hacer witness extraction. Capaz no es original lo nuestro (y capaz Coq lo banca con realizabilidad).

- Motivación, limitaciones de lógica clásica. Demostración sqrt 2
- Lógica intuicionista
- Como necesitamos reducir en ND, necesitamos la demo en ND. Escribirla en este caso.
- También queremos para $\Pi^0_2$, mostrar la extensión en ND.
- En realidad no nos sirve $\Gamma^{\neg\neg}$, queremos dejarlo como está y demostrar que los axiomas demuestran sus traducciones. Pero no vale siempre (buscar c.ej), caracterizar cuando.
- Sumarizar cómo queda, vincular con reducción. Mostrar ejemplos en PPA que funcionan y ejemplos que no.
- Extensión a demostraciones. Mostrar algunos ejemplos interesantes (y los que usen los lemas dNegRElim y rElim)
- Lemas para demostraciones: dNegRElim, rElim, tNegRElim
- Reducción (buena explicación https://plato.stanford.edu/entries/natural-deduction/). En realidad se conoce como **normalization**.
  - Similitud con reducción en cálculo lambda.
  - Ejemplos de LP y todo LPO
  - substHyp, substVar en proofs
  - Argumentos de que es correcto y completo?
  - Small step vs big step

Queremos, dado un teorema, *extraer testigos de un existencial*. Por ejemplo, si tenemos una demostración de $\exists x.p(x)$ la extracción nos debería instanciar $x$ en un término $t$ tal que $p(t)$. Imaginemos que tenemos el siguiente programa de PPA

```
axiom ax: p(v)
theorem thm: exists X . p(X)
proof
    take X := v
    thus p(v) by ax
end
```

¿Cómo hacemos para extraer La demostración generada por el certificador es **clásica**. La forma más fácil de extraer un testigo de una demostración es normalizarla y obtener el testigo de su forma normal. Pero esto no se puede hacer en general para lógica clásica, porque las demostraciones en general no son **constructivas**.

En la lógica clásica vale el *principio del tercero excluido*, comúnmente conocido por sus siglas en inglés, LEM (*law of excluded middle*).

**Prop. 1.** LEM Para toda fórmula $A$, es verdadera ella o su negación

$$A \vee \neg A$$

Las demostraciones que usan este principio suelen dejar aspectos sin concretizar, como muestra el siguiente ejemplo bien conocido:

**Teorema 2.** Existen dos números irracionales, $a, b$ tales que $a^b$ es irracional

*Demostración.* Considerar el número $\sqrt{2}^{\sqrt{2}}$. Por LEM, es o bien racional o irracional.

- Supongamos que es racional. Como sabemos que $\sqrt{2}$ es irracional, podemos tomar $a = b = \sqrt{2}$.
- Supongamos que es irracional. Tomamos $a = \sqrt{2}^{\sqrt{2}}, b = \sqrt{2}$. Ambos son irracionales, y tenemos

$$a^b = \left(\sqrt{2}^{\sqrt{2}}\right)^{\sqrt{2}} = \sqrt{2}^{\sqrt{2}\cdot\sqrt{2}} = \sqrt{2}^2 = 2,$$

que es racional.

□

Como se puede ver, la prueba no nos da forma de saber cuales son $a$ y $b$. Es por eso que en general, tener una demostración de un teorema que afirma la existencia de un objeto que cumpla cierta propiedad, no necesariamente nos da una forma de encontrar tal objeto. Entonces tampoco vamos a poder extraer un testigo.

En el caso de Teorema 2 , lo demostramos de una forma no constructiva pero existen formas constructivas de hacerlo (TODO: citar). Pero hay casos en donde no. Por ejemplo, si consideramos la fórmula

$$\exists x((x = 1 \wedge C) \vee (x = 0 \wedge \neg C))$$

y pensamos en $C$ como algo indecidible, por ejemplo `HALT`, trivialmente podemos demostrarlo de forma no constructiva (LEM con $C \vee \neg C$) pero no de forma constructiva.

## 5.1. Lógica intuicionista

Para solucionar estos problemas existe la lógica **intuicionista**, que se puede definir como la lógica clásica sin LEM. Al no contar con ese principio, las demostraciones son constructiva. Esto permite por un lado para tener interpretaciones computacionales (como la *BHK*) y además que exista la noción de *forma normal* de uan demostración. Existen métodos bien conocidos para reducir prueba hacia su forma normal con un proceso análogo a una reducción de cálculo $\lambda$. Luego en la forma normal se esperaría que toda demostración de un $\exists$ sea mediante I$\exists$, explicitando el testigo.

Al no tener LEM, tampoco valen principios equivalentes, como la eliminación de la doble negación (TODO: hablar un poco más de esto)

## 5.2. Traducción de Friedman

### 5.2.1. Traducción de doble negación

Queremos extraer testigos de las demostraciones generadas por el certificador de PPA, pero son en lógica clásica. Sabemos que podemos hacerlo para lógica intuicionista. ¿Cómo conciliamos ambos mundos?

Existen muchos métodos que permiten embeber la lógica clásica en la intuicionista (TODO: citar). Un mecanismo general es la traducción de **doble negación**, que intuitivamente consiste en agregar una doble negación recursivamente a toda la fórmula. Por ejemplo

**Def. 11.** (Traducción *Gödel-Gentzen*) Dada una fórmula $A$ se asocia con otra $A^{\mathrm{N}}$. La traducción se define inductivamente en la estructura de la fórmula de la siguiente forma

$$\begin{aligned}
\bot^{\mathrm{N}} &= \bot \\
A^{\mathrm{N}} &= \neg\neg A \quad \text{con } A \neq \bot \text{ atómica} \\
(A \wedge B)^{\mathrm{N}} &= A^{\mathrm{N}} \wedge B^{\mathrm{N}} \\
(A \vee B)^{\mathrm{N}} &= \neg(\neg A^{\mathrm{N}} \wedge \neg B^{\mathrm{N}}) \\
(A \to B)^{\mathrm{N}} &= A^{\mathrm{N}} \to B^{\mathrm{N}} \\
(\forall x.A)^{\mathrm{N}} &= \forall x.A^{\mathrm{N}} \\
(\exists x.A)^{\mathrm{N}} &= \neg\forall x.\neg A^{\mathrm{N}}
\end{aligned}$$

**Teorema 3.** Si tenemos $\vdash_C A$, luego $\vdash_I A^{\mathrm{N}}$

Esto significa que dada una demostración en lógica clásica, podemos obtener una en lógica intuicionista de su traducción. Pero esto no es exactamente lo que queremos, porque por ejemplo

$$(\exists x.p(x))^{\mathrm{N}} = \neg\forall x.\neg\neg\neg p(x)$$

Que al no ser una demostración de un $\exists$, al reducirla no necesariamente obtendremos un testigo.

### 5.2.2. El truco de Friedman

La idea de Friedman [Miq11] es generalizar la traducción Gödel-Gentzen reemplazando la negación intuicionista $\neg A \equiv A \to \bot$ por una relativa $\neg_R A \equiv A \to R$ que está parametrizada por una fórmula arbitraria $R$. Esto nos va a permitir, con una elección particular de $R$, traducir una demostración clásica de una fórmula $\Sigma^0_1$ (e incluso $\Pi^0_2$) a una intuicionista, y usarla para demostrar **la fórmula original**. Finalmente podremos reducirla y hacer la extracción de forma usual.

**Def. 12.** (Traducción de doble negación relativizada)

$$\begin{aligned}
\bot^{\neg\neg} &= \bot \\
A^{\neg\neg} &= \neg_R\neg_R A \quad \text{con } A \neq \bot \text{ atómica} \\
(A \wedge B)^{\neg\neg} &= A^{\neg\neg} \wedge B^{\neg\neg} \\
(A \vee B)^{\neg\neg} &= \neg_R(\neg_R A^{\neg\neg} \wedge \neg_R B^{\neg\neg}) \\
(A \to B)^{\neg\neg} &= A^{\neg\neg} \to B^{\neg\neg} \\
(\forall x.A)^{\neg\neg} &= \forall x.A^{\neg\neg} \\
(\exists x.A)^{\neg\neg} &= \neg_R\forall x.\neg_R A^{\neg\neg}
\end{aligned}$$

**Teorema 4.** Si $\Gamma \vdash_C A$, luego $\Gamma^{\neg\neg} \vdash_I A^{\neg\neg}$

Veremos esta extensión de la traducción a contextos y demostraciones más adelante.

Veamos cómo podemos usarla para, dada una demostración clásica de $\exists x A$ obtener una intuicionista.

**Prop. 2.** Sea $\Pi$ una demostración clásica de $\exists x.A$, y $A$ una fórmula atómica. Si tenemos

$$\Gamma \vdash_C \exists x.A,$$

luego

$$\Gamma^{\neg\neg} \vdash_I \exists x.A.$$

*Demostración.* Aplicando la traducción, tenemos que

$$\frac{\Pi}{\Gamma \vdash_C \exists x.A}$$

se traduce a

$$\frac{\Pi^{\neg\neg}}{\Gamma^{\neg\neg} \vdash_I \neg_R\forall x.\neg_R\neg_R\neg_R A}$$

luego, tomando $R$ como la fórmula que queremos probar, $\exists x.A$

$$\begin{aligned}
\Pi^{\neg\neg} \rhd \Gamma^{\neg\neg} &\vdash_I \neg_R\forall x.\neg_R\neg_R\neg_R A \\
&\iff \Gamma^{\neg\neg} \vdash_I \neg_R\forall x.\neg_R A && (1) \\
&= \Gamma^{\neg\neg} \vdash_I (\forall x.(A \to R)) \to R \\
&= \Gamma^{\neg\neg} \vdash_I (\forall x.(A \to \exists x.A)) \to \exists x.A && (R = \exists x A) \\
&\Rightarrow \Gamma^{\neg\neg} \vdash_I \exists x.A && (1)
\end{aligned}$$

□

**Lema 1.** $\neg_R\neg_R\neg_R A \iff \neg_R A$

*Demostración.* (TODO: En deducción natural) □

**Obs. 1.** $\vdash_I \forall x(A \to \exists x A)$

(TODO: IDem pero en ND, y también para $\forall$)

# 6. LA HERRAMIENTA PPA

```haskell
type VarId = String
type FunId = String
type PredId = String
type HypId = String

data Term
    = TVar VarId
    | TMetavar Metavar
    | TFun FunId [Term]
```

```haskell
data Form
    = FPred PredId [Term]
    | FAnd Form Form
    | FOr Form Form
    | FImp Form Form
    | FNot Form
    | FTrue
    | FFalse
    | FForall VarId Form
    | FExists VarId Form
```

*Fig. 6.1:* Modelado de fórmulas y términos de LPO

Las meta-variables se usan para unificación, que es parte del solver de PPA. Ver más en Subsección 4.3.7 Unificación

```
data Proof =
    | PAx HypId
    | PAndI
        { proofLeft :: Proof
        , proofRight :: Proof
        }
    | PAndE1
        { right :: Form
        , proofAnd :: Proof
        }
    | PAndE2
        { left :: Form
        , proofAnd :: Proof
        }
    | POrI1
        { proofLeft :: Proof
        }
    | POrI2
        { proofRight :: Proof
        }
    | POrE
        { left :: Form
        , right :: Form
        , proofOr :: Proof
        , hypLeft :: HypId
        , proofAssumingLeft :: Proof
        , hypRight :: HypId
        , proofAssumingRight :: Proof
        }
    | PImpI
        { hypAntecedent :: HypId
        , proofConsequent :: Proof
        }
    | PImpE
        { antecedent :: Form
        , proofImp :: Proof
        , proofAntecedent :: Proof
        }
    | PNotI
        { hyp :: HypId
        , proofBot :: Proof
        }
    | PNotE
        { form :: Form
        , proofNotForm :: Proof
        , proofForm :: Proof
        }
    | PTrueI
    | PFalseE
        { proofBot :: Proof
        }
    | PLEM
    | PForallI
        { newVar :: VarId
        , proofForm :: Proof
        }
    | PForallE
        { var :: VarId
        , form :: Form
        , proofForall :: Proof
        , termReplace :: Term
        }
    | PExistsI
        { inst :: Term
        , proofFormWithInst :: Proof
        }
    | PExistsE
        { var :: VarId
        , form :: Form
        , proofExists :: Proof
        , hyp :: HypId
        , proofAssuming :: Proof
        }
```

*Fig. 6.2:* Modelado de reglas de inferencia para demostraciones

El modelado de las reglas de inferencia omite varios detalles que están implícitos y serán inferidos por el algoritmo de chequeo. De esa forma las demostraciones son más fáciles de escribir y generar. Por ejemplo, `PImpI` no especifica en su modelo cuál es la implicación que se está introduciendo, dado que durante el chequeo debería ser la fórmula actual a demostrar

## 6.1. Compiladores

- Primer de compiladores en general y sus frontends
- Parser generators en general. LR/LALR
- Happy. Alex.
- Sintaxis BNF en apéndice (el archivo). Incluir el archivo Alex/happy? Es cortito

# 7. CONCLUSIONES

- (TODO: Al final de la tesis)

Trabajo futuro

- Inducción como axioma de deducción natural, permite demostrar más cosas
- Sofisticación del by para eliminación de foralls más compleja que consecutivos. (elimina un solo forall, pero no intenta recursivamente eliminar todos)
- Sofisticación de PPA como lenguaje de programación. Hacer una standard library de teoremas. Permitir importar archivos/módulos. Proveer teorías by default importables.
- Mejorar la eliminación de testigos?

# BIBLIOGRAFíA


[BW05] Henk Barendregt y Freek Wiedijk. «The challenge of computer mathematics». En: *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 363.1835 (sep. de 2005), págs. 2351-2375. ISSN: 1471-2962. DOI: 10.1098/rsta.2005.1650. URL: http://dx.doi.org/10.1098/rsta.2005.1650.

[Gen35] Gerhard Gentzen. «Untersuchungen über das logische Schließen. I». En: *Mathematische Zeitschrift* 39.1 (dic. de 1935), págs. 176-210. ISSN: 1432-1823. DOI: 10.1007/BF01201353. URL: https://doi.org/10.1007/BF01201353.

[Miq11] Alexandre Miquel. «Existential witness extraction in classical realizability and via a negative translation». En: *Log. Methods Comput. Sci.* 7.2 (2011). DOI: 10.2168/LMCS-7(2:2)2011. URL: https://doi.org/10.2168/LMCS-7(2:2)2011.

[Wie] Freek Wiedijk. *Mathematical Vernacular*. https://www.cs.ru.nl/~freek/notes/mv.pdf.