



UNIVERSIDAD DE BUENOS AIRES
FACULTAD DE CIENCIAS EXACTAS Y NATURALES
DEPARTAMENTO DE COMPUTACIÓN

PPA

Un asistente de demostraciones tipo Mizar para lógica clásica de primer orden, con extracción de testigos basada en la traducción de Friedman

Tesis de Licenciatura en Ciencias de la Computación

Manuel Panichelli

Director: Pablo Barenbaum
Buenos Aires, 2024

PPA: UN ASISTENTE DE DEMOSTRACIONES TIPO MIZAR PARA LÓGICA CLÁSICA DE PRIMER ORDEN CON EXTRACCIÓN DE TESTIGOS BASADA EN 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 MIZAR-LIKE PROOF-ASSISTANT FOR CLASSICAL FIRST-ORDER LOGIC WITH WITNESS EXTRACTION BASED ON 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 |
| 1.2.1. | Mizar | 2 |
| 1.3. | Arquitectura de PPA | 2 |
| 1.4. | Lógica de primer orden | 3 |
| 2.. | Deducción natural | 4 |
| 2.1. | Deducción natural | 6 |
| 2.1.1. | Reglas de inferencia | 8 |
| 2.1.2. | Hipótesis nombradas | 10 |
| 2.1.3. | Meta-teoremas | 10 |
| 2.2. | Algoritmos | 11 |
| 2.2.1. | Chequeo | 13 |
| 2.2.2. | Alpha equivalencia | 13 |
| 2.2.3. | Sustitución sin capturas | 14 |
| 2.2.4. | Variables libres | 15 |
| 3.. | PPA | 17 |
| 3.1. | Interfaz | 22 |
| 3.1.1. | Fórmulas | 23 |
| 3.1.2. | Demostraciones | 23 |
| 3.2. | Compilador | 29 |
| 3.3. | Certificador | 29 |
| 3.3.1. | Unificación | 29 |
| 4.. | Extracción de testigos de existenciales | 30 |
| 4.1. | Lógica intuicionista | 32 |
| 4.2. | Traducción de Friedman | 33 |
| 4.2.1. | Traducción de doble negación | 33 |
| 4.2.2. | El truco de Friedman | 33 |
| 5.. | Conclusiones | 35 |

1. INTRODUCCIÓN

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 demostracion. 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

1.2.1. Mizar

(TODO: Mizar)

1.3. Arquitectura de PPA

- Por qué certificados (criterio de De Bruijn)
- 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

1.4. Lógica de primer orden

Def. 1. Términos Un término puede ser

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

Def. 2. Fórmulas Una fórmula puede ser

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

(TODO: Definiciones, repaso, lo necesario)

2. DEDUCCIÓN NATURAL

- Ejemplo de demostración en lenguaje natural ✓
- Necesitamos una forma estructural de representar demostraciones ✓
- Proof calculus / proof system enmarcado en Proof theory. Cómo están compuestos en general ✓
- Natural deduction ✓
- Reglas de introducción y eliminación ✓
- Formalización del ejemplo ✓
- Ejemplo con cuantificadores
- Cut como meta-teorema (y meta teoremas en general)
- Implementación de data types principales ✓
- Algoritmo de chequeo
- Algoritmos adicionales: alpha igualdad, variables libres, sust sin capturas ✓

Vamos a comenzar por las fundaciones: Queremos armar un programa que permita escribir teoremas y demostraciones. ¿Cómo se representa una demostración en la computadora? 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 del documento. 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.
- Sabemos que si reprueba, entonces recursa.
- \therefore recursa.

□

¿Cómo podemos formalizarla en un *proof system*? En general, van a incluir los siguientes componentes

- **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 $A \rightarrow B$ y A , se puede concluir B) o *modus tollens* (si es cierto $A \rightarrow B$ 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): $A \vee \neg A$

2.1. Deducción natural

El sistema 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 , \dots), que vamos a introducir con un ejemplo.

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

Definiciones preliminares:

(TODO: Agregar estas definiciones)

- Relación \vdash
- Contexto Γ
- FVs de un contexto
- Sustitución sin capturas

$$\begin{array}{c}
\frac{\Gamma \vdash \perp}{\Gamma \vdash A} E_{\perp} \qquad \frac{}{\Gamma \vdash \top} I_{\top} \\
\frac{}{\Gamma \vdash A \vee \neg A} LEM \qquad \frac{}{\Gamma, h : A \vdash h : A} Ax \\
\\
\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \wedge B} I_{\wedge} \\
\frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A} E_{\wedge_1} \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash B} E_{\wedge_2} \\
\\
\frac{\Gamma \vdash A}{\Gamma \vdash A \vee B} I_{\vee_1} \qquad \frac{\Gamma \vdash B}{\Gamma \vdash A \vee B} I_{\vee_2} \\
\frac{\Gamma \vdash A \vee B \quad \Gamma, A \vdash C \quad \Gamma, B \vdash C}{\Gamma \vdash C} E_{\vee} \\
\\
\frac{\Gamma, h : A \vdash B}{\Gamma \vdash A \rightarrow B} I_{\rightarrow} \qquad \frac{\Gamma \vdash A \rightarrow B \quad \Gamma \vdash A}{\Gamma \vdash B} E_{\rightarrow} \\
\frac{\Gamma, h : A \vdash \perp}{\Gamma \vdash \neg A} I_{\neg} \qquad \frac{\Gamma \vdash \neg A \quad \Gamma \vdash A}{\Gamma \vdash \perp} E_{\neg} \\
\\
\frac{\Gamma \vdash A \quad x \notin fv(\Gamma)}{\Gamma \vdash \forall x.A} I_{\forall} \qquad \frac{\Gamma \vdash \forall x.A}{\Gamma \vdash A\{x := t\}} E_{\forall} \\
\\
\frac{\Gamma \vdash A\{x := t\}}{\Gamma \vdash \exists x.A} I_{\exists} \\
\frac{\Gamma \vdash \exists x.A \quad \Gamma, A \vdash B \quad x \notin fv(\Gamma, B)}{\Gamma \vdash B} E_{\exists}
\end{array}$$

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

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

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

Queremos probar entonces

$$\left((X \rightarrow R) \wedge (F \rightarrow X) \right) \rightarrow (F \rightarrow R)$$

$$\begin{array}{c}
\frac{\Gamma \vdash (X \rightarrow R) \wedge (F \rightarrow X)}{\Gamma \vdash X \rightarrow R} \text{Ax} \quad \frac{\Gamma \vdash (X \rightarrow R) \wedge (F \rightarrow X)}{\Gamma \vdash F \rightarrow X} \text{E}\wedge_2 \quad \frac{}{\Gamma \vdash F} \text{Ax} \\
\frac{}{\Gamma \vdash X \rightarrow R} \text{E}\wedge_1 \quad \frac{\Gamma \vdash F \rightarrow X}{\Gamma \vdash X} \text{E}\rightarrow \\
\frac{\Gamma = h_1 : (X \rightarrow R) \wedge (F \rightarrow X), h_2 : F \vdash R}{h_1 : (X \rightarrow R) \wedge (F \rightarrow X) \vdash F \rightarrow R} \text{I}\rightarrow \\
\frac{h_1 : (X \rightarrow R) \wedge (F \rightarrow X) \vdash F \rightarrow R}{\vdash ((X \rightarrow R) \wedge (F \rightarrow X)) \rightarrow (F \rightarrow R)} \text{I}\rightarrow
\end{array}$$

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

Paso por paso,

- $\text{I}\rightarrow$: *introducimos* la implicación. Para demostrarla, asumimos el antecedente y en base a eso demostramos el consecuente. Es decir asumimos $(X \rightarrow R) \wedge (F \rightarrow X)$, y en base a eso queremos deducir $F \rightarrow R$. Las *hipótesis* están etiquetadas, en este caso h_1 .
- $\text{I}\rightarrow$: Asumimos F , nos queda probar R . Renombramos el *contexto* de hipótesis como Γ .
- La estrategia para probar R es usando la siguiente cadena de implicaciones: $F \rightarrow X \rightarrow R$, y sabemos que vale F . Como tenemos que probar R , arrancamos de atrás para adelante.
- $\text{E}\rightarrow$: *eliminamos* una implicación, la usamos para deducir su conclusión demostrando el antecedente. Esta regla de inferencia tiene dos partes, probar la implicación $(X \rightarrow R)$, y probar el antecedente (X) .
 - Para probar la implicación, tenemos que usar la hipótesis h_1 , *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 raíces del árbol, los casos base, suelen ser aplicaciones de la regla de inferencia Ax , que permite deducir fórmulas citando hipótesis del contexto.

2.1.1. Reglas de inferencia

A continuación se explican brevemente las reglas de inferencia listadas en [Figura 2.1 Reglas de inferencia para deducción natural de lógica de primer orden](#).

Reglas base

$$\begin{array}{c}
\frac{\Gamma \vdash \perp}{\Gamma \vdash A} \text{E}\perp \quad \frac{}{\Gamma \vdash \top} \text{I}\top \\
\frac{}{\Gamma \vdash A \vee \neg A} \text{LEM} \quad \frac{}{\Gamma, h : A \vdash h : A} \text{Ax}
\end{array}$$

- $\text{E}\perp$: A partir de \perp , algo que es falso, vamos a poder deducir cualquier fórmula.

- IT: \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.

Reglas de conjunciones y disyunciones

$$\begin{array}{c}
 \frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \wedge B} I\wedge \\
 \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A} E\wedge_1 \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash B} E\wedge_2 \\
 \frac{\Gamma \vdash A}{\Gamma \vdash A \vee B} I\vee_1 \qquad \frac{\Gamma \vdash B}{\Gamma \vdash A \vee B} I\vee_2 \\
 \frac{\Gamma \vdash A \vee B \quad \Gamma, A \vdash C \quad \Gamma, B \vdash C}{\Gamma \vdash C} E\vee
 \end{array}$$

- $I\wedge$: 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.

Reglas de implicación y negación

$$\begin{array}{c}
 \frac{\Gamma, h : A \vdash B}{\Gamma \vdash A \rightarrow B} I\rightarrow \qquad \frac{\Gamma \vdash A \rightarrow B \quad \Gamma \vdash A}{\Gamma \vdash B} E\rightarrow \\
 \frac{\Gamma, h : A \vdash \perp}{\Gamma \vdash \neg A} I\neg \qquad \frac{\Gamma \vdash \neg A \quad \Gamma \vdash A}{\Gamma \vdash \perp} E\neg
 \end{array}$$

- $I\rightarrow$: Para demostrar una implicación, asumimos el antecedente (agregándolo a las hipótesis) y en base a eso se demuestra el consecuente.
- $E\rightarrow$: 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.

Reglas de cuantificadores

(DUDA: Validar las justificaciones coloquiales de acá)

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 un \exists una disyunción.

$$\frac{\Gamma \vdash A \quad x \notin fv(\Gamma)}{\Gamma \vdash \forall x.A} \text{I}\forall$$

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

- $\text{I}\forall$: 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 Γ no tengo que tenerlo ligado a nada, sino no lo estaría demostrando en general.
- $\text{E}\forall$: Para usar un $\forall x.A$ para demostrar, como vale para todo x , puedo instanciarlo en *cualquier término* t .

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

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

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

Ejemplo 3. (TODO: Ejemplo con cuantificadores, el mismo de alumnos pero con forall)

2.1.2. Hipótesis nombradas

(TODO: presentamos contextos como conjuntos y reglas de hypids sin etiquetar, pero en realidad deberían estar etiquetadas en rigor.)

2.1.3. Meta-teoremas

Antes mencionamos *modus tollens* como regla de inferencia, pero no lo listamos anteriormente. 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. Nos va a servir para simplificar el resto de PPA, dado que vamos a generar demostraciones en deducción natural. Estas reglas de inferencia las demostramos como **meta-teoremas**

(TODO: Demo modus tollens)

(TODO: Cut como meta teorema)

(TODO: DnegElim como meta teorema, y cómo permite razonar por el absurdo)

2.2. Algoritmos

Vamos a hablar de los algoritmos que se implementaron sobre deducción natural. Pero primero presentamos el modelo de fórmulas y demostraciones, que es central a todo el programa.

```

type VarId = String
type FunId = String
type PredId = String
type HypId = String

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

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. 2.3: 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 3.3.1 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. 2.4: 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

2.2.1. Chequeo

(TODO: Completar sección. Cómo representamos el algoritmo matemáticamente?)

2.2.2. 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. 3. Alpha equivalencia. Se define la relación $\stackrel{\alpha}{\equiv}$ inductivamente sobre la estructura de la fórmula

- Términos

$$\begin{aligned} x &\stackrel{\alpha}{\equiv} y \iff x = y \\ f(t_1, \dots, t_n) &\stackrel{\alpha}{\equiv} f(u_1, \dots, u_n) \iff f = g \wedge t_1 \stackrel{\alpha}{\equiv} u_1 \wedge \dots \wedge t_n \stackrel{\alpha}{\equiv} u_n \end{aligned}$$

Caso contrario, no son equivalentes (si usan símbolos de función diferentes o comparamos diferentes estructuras, como funciones con variables o de distinta aridad)
(DUDA: mejor forma de escribir esto?)

- Fórmulas

$$\begin{aligned} \perp &\stackrel{\alpha}{\equiv} \perp \\ \top &\stackrel{\alpha}{\equiv} \top \\ p(t_1, \dots, t_n) &\stackrel{\alpha}{\equiv} p(u_1, \dots, u_n) \iff t_1 \stackrel{\alpha}{\equiv} u_1 \wedge \dots \wedge t_n \stackrel{\alpha}{\equiv} u_n \\ (A \wedge B) &\stackrel{\alpha}{\equiv} (A' \wedge B') \iff A \stackrel{\alpha}{\equiv} A' \wedge B \stackrel{\alpha}{\equiv} B' \\ (A \vee B) &\stackrel{\alpha}{\equiv} (A' \vee B') \iff A \stackrel{\alpha}{\equiv} A' \wedge B \stackrel{\alpha}{\equiv} B' \\ (A \rightarrow B) &\stackrel{\alpha}{\equiv} (A' \rightarrow B') \iff A \stackrel{\alpha}{\equiv} A' \wedge B \stackrel{\alpha}{\equiv} B' \\ (\neg A) &\stackrel{\alpha}{\equiv} (\neg A') \iff A \stackrel{\alpha}{\equiv} A' \\ (\forall x.A) &\stackrel{\alpha}{\equiv} (\forall y.A') \iff A\{x := z\} \stackrel{\alpha}{\equiv} A'\{y := z\} \text{ con } z \text{ fresca} \\ (\exists x.A) &\stackrel{\alpha}{\equiv} (\exists y.A') \iff A\{x := z\} \stackrel{\alpha}{\equiv} A'\{y := z\} \text{ con } z \text{ fresca} \end{aligned}$$

Caso contrario, no son equivalentes.

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, insertamos en cada sustitución

los mapeos $x \mapsto z$ y $y \mapsto z$. Luego, cuando estemos comparando dos variables libres, chequeamos que *mapeen* a variables iguales. En este ejemplo son alpha equivalentes, pues

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

2.2.3. 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 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.

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

donde z es una variable *fresca*.

Def. 4. Sustitución sin capturas. Se define inductivamente en la estructura la fórmula. Sean y una variable y t un término cualquiera.

■ Términos

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

■ Fórmulas

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

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.

2.2.4. Variables libres

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

Def. 5. Variables libres de fórmulas y términos. Se definen inductivamente en su estructura de la siguiente forma.

■ Términos

$$\begin{aligned}
& fv(x) = \{x\} \\
& fv(f(t_1, \dots, t_n)) = \bigcup_{i \in 1 \dots n} fv(t_i)
\end{aligned}$$

■ Fórmulas

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

Def. 6. Variables libres de un contexto

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

Def. 7. Variables libres de una demostración. Sea Π una demostración. $fv(\Pi)$ son las variables libres de todas las fórmulas que la componen. Por ejemplo, para la siguiente

$$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \wedge B} \text{I}\wedge$$

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

3. 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. Considerar incluir la de grupos
- Compiladores
 - Primer de compiladores en general y sus frontends
 - Parser generators en general. LR/LALR
 - Happy. Alex.
 - Sintaxis EBNF. Incluir el archivo Alex/happy? Es cortito
- Certificador: componente de PPA que certifica 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 forall 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.

PPA (*Pani's Proof Assistant*) se construye sobre las fundaciones de deducción natural. Es un *proof assistant* que permite escribir de una forma práctica demostraciones de cualquier teoría de lógica clásica de primer orden. Para introducir el lenguaje veamos un ejemplo, luego se detalla la interfaz y finalmente veremos los detalles de implementación. Representamos el mismo de alumnos del [Ejemplo 1](#) pero esta vez en primer orden y con un poco más de sofisticación.

```

1 axiom reprueba_recu_parcial_recura: 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_recura:
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_recura
35 end
36
37 theorem falta_recu_recura:
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_recura
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_recura`: 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_recura: 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_recura`, 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.

```

1 theorem reprueba_recu_recura:
2   forall A. forall M.
3     reprueba(A, recu(parcial(M))) -> recursa(A, M)
4 proof
5   let A
6   let M
7   suppose reprueba_recu: reprueba(A, recu(parcial(M)))
8
9   claim reprueba_p: reprueba(A, parcial(M))
10  proof
11    have rinde_recu: rinde(A, recu(parcial(M)))
12    by reprueba_recu, reprobo_rinde
13
14    hence reprueba(A, parcial(M))
15    by rinde_recu_reprobo_parcial
16  end
17
18  hence recursa(A, M)
19  by reprueba_recu,
20    reprueba_p,
21    reprueba_recu_parcial_recura
22 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.

```

1 theorem falta_recu_recura:
2   forall A. forall M.
3     falta(A, recu(parcial(M))) -> recursa(A, M)
4 proof
5   let A'
6   let M'
7
8   suppose falta_recu: falta(A', recu(parcial(M')))
9
10  have reprueba_recu: reprueba(A', recu(parcial(M')))
11    by falta_recu, falta_reprueba
12
13  hence recursa(A', M') by reprueba_recu_recura
14 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 de programación 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. Será explicado más adelante.

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

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

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

```

1 theorem <name> : <form>
2 proof
3   <steps>
4 end

```

Los identificadores se dividen en tres tipos:

- **Variables** (<var>)

$(\backslash_|[A-Z])[a-zA-Z0-9\backslash_]*(')*$

- **Identificadores** (<id>)

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

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

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

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

3.1.1. 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** `~` <form>

- **Cuantificadores**

- `exists` <var>. <form>
- `forall` <var>. <form>

- `true`, `false`

- **Paréntesis:** (<form>)

3.1.2. 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. Mapean aproximadamente a reglas de inferencia de deducción natural.

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.

by

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, que es completo para lógica proposicional pero heurístico para primer orden (elimina todos los forall 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 [Sección 3.1.2 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

```

1 axiom ax1: a -> b
2 axiom ax2: b -> c
3
4 theorem t1: a -> c
5 proof
6   suppose a: a
7
8   // La tesis ahora es c
9   thus c by a, ax1, ax2
10 end
```

Y para eliminación de cuantificadores universales

```

1 axiom ax: forall X . f(X)
2
3 theorem t: f(n)
4 proof
5   thus f(n) by ax
6 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.

```

1 axiom ax1: a -> b
2 axiom ax2: b -> c
3
4 theorem t1: a -> c
5 proof
6   suppose a: a
7   have b: b by a, ax1
```

```

8      thus c by b, ax2
9  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,

```

1  axiom ax1: a -> b
2  axiom ax2: b -> c
3
4  theorem t1: a -> c
5  proof
6      suppose a: a
7      have b: b by a, ax1
8      thus c by b, ax2
9  end

```

```

10 theorem t1': a -> c
11 proof
12     suppose a: a
13     have b: b by -, ax1
14     thus c by -, ax2
15 end
16
17 theorem t1'': a -> c
18 proof
19     suppose -: a
20     then -: b by ax1
21     hence c by ax2
22 end

```

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

```

1  theorem "distributiva de negación sobre disyunción":
2      ~(a | b) <-> ~a & ~b
3  proof
4      thus ~(a | b) <-> ~a & ~b
5  end

```

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 |
|----------------|-----------------|
| $I\exists$ | take |
| $E\exists$ | consider |
| $I\forall$ | let |
| $E\forall$ | by |
| $I\forall_1$ | by |
| $I\forall_2$ | by |
| $E\vee$ | cases |
| $I\wedge$ | by |
| $E\wedge_1$ | by |
| $E\wedge_2$ | by |
| $I\rightarrow$ | suppose |
| $E\rightarrow$ | by |
| $I\neg$ | suppose |
| $E\neg$ | by |
| IT | by |
| $E\perp$ | by |
| LEM | by |
| Ax | by |

Tab. 3.1: Reglas de inferencia y comandos

■ **suppose** ($I\rightarrow$ / $I\neg$)

Si la tesis es una implicación $A \rightarrow B$, agrega el antecedente A como hipótesis con el nombre dado y reduce la tesis al consecuente B . Viendo la negación como una implicación $\neg A \equiv A \rightarrow \perp$, se puede usar para razonar por el absurdo, tomando $B = \perp$.

```

1 theorem "suppose":
2   a -> (a -> b) -> b
3 proof
4   suppose h1: a
5   suppose h2: a -> b
6   thus b by h1, h2
7 end
```

```

1 theorem "not intro":
2   ~b & (a -> b) -> ~a
3 proof
4   suppose h: ~b & (a -> b)
5   suppose a: a
6   hence false by h, a
7 end
```

■ **cases** ($E\vee$)

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

```

1 theorem "cases":
2   (a & b) | (c & a) -> a
3 proof
4   suppose h: (a & b) | (c & a)
5   cases by h
6     case a & b
7       hence a
8     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 $c1 \mid \dots \mid cm$ by $h1, \dots, hn$.

Por lo que en el ejemplo anterior, podríamos haber usado el mismo case incluso si la hipótesis fuera $\sim((\sim a \mid \sim b) \ \& \ (\sim c \mid \sim a))$, pues es equivalente a $(a \ \& \ b) \mid (c \ \& \ a)$

■ **take** (\exists)

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** (\exists)

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 α -equivalente, por ejemplo si podemos justificar **exists** X . $p(X)$, podemos usarlo para **consider** Y **st** h : $p(Y)$ **by** ...

■ **let** (\forall)

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.

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

```

1  theorem "and discharge" : a -> b -> (a & b)
2  proof
3      suppose "a" : a
4      suppose "b" : b
5      // La tesis es a & b
6      hence b by "b"
7
8      // La tesis es a
9      thus a by "a"
10 end

```

Esto puede ser prácticamente en cualquier orden.

Fig. 3.3: Descarga de conjunción compleja

```

1  axiom "a": a
2  axiom "b": b
3  axiom "c": c
4  axiom "d": d
5  axiom "e": e
6  theorem "and discharge" : (a & b) & ((c & d) & e)
7  proof
8      thus a & e by "a", "e"
9      thus d by "d"
10     thus b & c by "b", "c"
11 end

```

Otros comandos

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

```

1  axiom a1: ~a
2  axiom a2: ~b
3
4  theorem "ejemplo" : ~(a | b)
5  proof
6      equivalently ~a & ~b
7      thus ~a by a1
8      thus ~b by a2
9  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

3.2. Compilador

3.3. Certificador

3.3.1. Unificación

4. EXTRACCIÓN DE TESTIGOS DE EXISTENCIALES

Puntos a abordar

- 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 Π_2^0 , 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 1. 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 1**, 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.

4.1. 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 una demostración. Existen métodos bien conocidos para reducir prueba hacia su forma normal con un proceso análogo a una reducción de cálculo λ . 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**)

4.2. Traducción de Friedman

4.2.1. 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. 8. (Traducción *Gödel-Gentzen*) Dada una fórmula A se asocia con otra A^N . La traducción se define inductivamente en la estructura de la fórmula de la siguiente forma

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

Teorema 2. Si tenemos $\vdash_C A$, luego $\vdash_I A^N$

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

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

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

4.2.2. El truco de Friedman

La idea de Friedman [Miq11] es generalizar la traducción Gödel-Gentzen reemplazando la negación intuicionista $\neg A \equiv A \rightarrow \perp$ por una relativa $\neg_R A \equiv A \rightarrow R$ 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 Σ_1^0 (e incluso Π_2^0) a una intuicionista, y usarla para demostrar **la fórmula original**. Finalmente podremos reducirla y hacer la extracción de forma usual.

Def. 9. (Traducción de doble negación relativizada)

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

Teorema 3. 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 $\exists xA$ obtener una intuicionista.

Prop. 2. Sea Π 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

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

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

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

□

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

Demostración. (TODO: En deducción natural)

□

Obs. 1. $\vdash_I \forall x(A \rightarrow \exists xA)$

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

5. 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 forall 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

- [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](https://doi.org/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](https://doi.org/10.2168/LMCS-7(2:2)2011). URL: [https://doi.org/10.2168/LMCS-7\(2:2\)2011](https://doi.org/10.2168/LMCS-7(2:2)2011).
- [Wie] Freek Wiedijk. *Mathematical Vernacular*. <https://www.cs.ru.nl/~freek/notes/mv.pdf>.